PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04L — A2

(11) International Publication Number: **WO 97/28623**

(43) International Publication Date: 7 August 1997 (07.08.97)

(21) International Application Number: PCT/IL97/00022

(22) International Filing Date: 15 January 1997 (15.01.97)

(30) Priority Data:
116804 17 January 1996 (17.01.96) IL

(71) Applicant *(for all designated States except US)*: MENTA SOFTWARE LTD. [IL/IL]; 3 Haplada Street, 60218 Or Yehuda (IL).

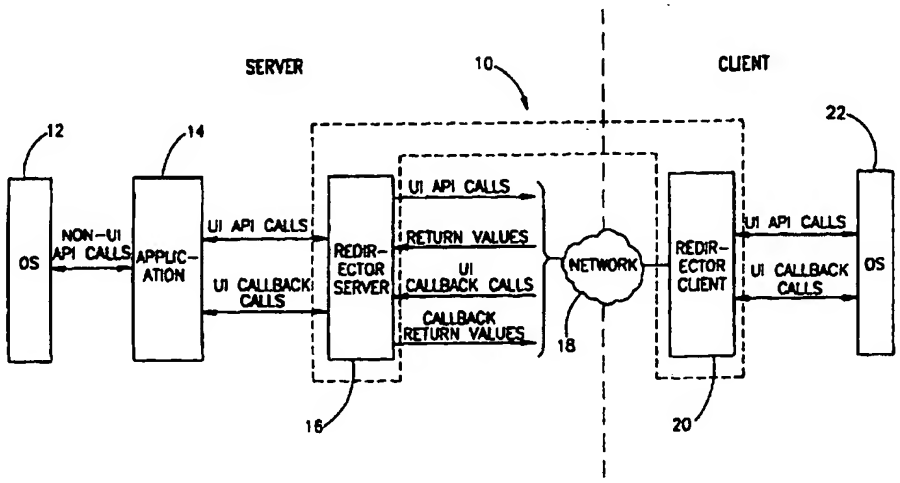
(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: GOLAN, Gilad [IL/IL]; 103 Usishkin Street, 47210 Ramat Hasharon (IL). ZANGVIL, Avner [IL/IL]; 49 Sheinkin Street, 65233 Tel Aviv (IL). ZANGVIL, Arnon [IL/IL]; 49 Sheinkin Street, 65233 Tel Aviv (IL).

(74) Agent: A. TALLY EITAN - ZEEV PEARL, D. LATZER & CO.; Law Offices, Lumir House, 22 Maskit Street, 46733 Herzelia (IL).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: APPLICATION USER INTERFACE REDIRECTOR



(57) Abstract

A novel application user interface redirector is disclosed that operates to extend an operating system, like Windows 95 or Windows NT, to allow applications to be used on one machine while actually executing on another machine. Most elements of the application execute on the server while the user interface elements of the application execute on the client. The result is that applications perform most operations, including input/output (I/O) intensive and CPU intensive operations, on the server but interact with the user on the local machine like any local application would. Multi-user capabilities are extended to support execution of applications, supporting multiple concurrent remote users. Utilizing the present invention, applications can execute on mixed architectures. Performance is further improved by providing a user interface skeleton or virtual user interface that locally generates the calls to the callback functions of the application. Any callback function return values are passed to the client. This improves the performance of the application user interface redirector over slow connections such as modems over dial up lines.

APPLICATION USER INTERFACE REDIRECTOR

FIELD OF THE INVENTION

The present invention relates to a system for redirecting the user interface of an application executing on a graphical user interface (GUI) operating system (OS).

BACKGROUND OF THE INVENTION

Currently, the remote access market is exploding, being driven by the need to extend strategic networking services to a fast growing base of mobile users, remote branches, and telecommuters. Remote access, once confined to occasional use by traveling executives, has become an integral part of mainstream networking services, and network administrators are faced with the challenge of implementing remote access solutions that are fast, easy to use, reliable, manageable, scaleable and secure. Current solutions successfully extend many network services and resources to remote users, but fail to meet the need to remotely execute local area network (LAN) based strategic applications.

In recent years, the enterprise network has become the lifeblood of corporate information systems, supporting every aspect of corporate activity. The necessity of making the network services and applications available to all corporate users, combined with a fast growing base of remote users, are driving the remote access market. The remote access market consists of three segments, all experiencing fast growth: telecommuters, mobile users and remote offices.

Telecommuting has become an accepted way of conducting business in the United States (US), due to the benefits to both employees and employers. Nine million US employees work from home at least part of the week, and it is believed that one fourth of the US work force will telecommute by the year 2000. To make telecommuting effective, network services must be extended to provide telecommuters with strategic corporate applications and workgroup applications that make up for distances and lack of contact with their colleagues.

A growing base of mobile sales, service employees and outside contractors need to be provided with the same set of strategic applications and resources that are available on

the corporate LAN in order to remain competitive. They have the need for high speed access to LAN based applications from hotels and customers sites, connecting through telephone lines.

Competitive and fast moving markets forced large organizations to decentralize, delegating responsibility to remote offices and branches in order to expedite decision making and better respond to customer needs. Many remote offices have only a handful of employees and no on-site systems expertise. In order for these companies to remain competitive, they need to expand strategic LAN based applications to their remote branches and offices.

A remote access solution known in the art is a remote node connection. Remote node enables users to establish a LAN node by dialing in from a remote location and access LAN based files and resources. Remote node was adopted by network administrators because it offered a manageable solution for large scale deployment and was enabled by high speed connection technologies such as V.34 modems and Integrated Services Digital Network (ISDN). The Windows 95 operating system includes a built in remote node client, and the Windows NT operating system includes both a remote node client and server.

Remote node is suitable for accessing LAN based files, facsimiles and printers. However, it is limited by the low bandwidth typically used to connect the remote node to the LAN. This limitation makes running LAN based or client/server applications prohibitively slow. Many client/server applications were not designed for use over low bandwidth connections and running them over phone connections is impractical. Similarly, launching an application that is installed on the LAN also results in unacceptable delay at the remote site because huge executable files are required to be transferred over the slow remote connection.

The bandwidth bottleneck limiting the functionality of remote node is not going to disappear. Bandwidth available to remote users range from 400 times slower than LAN's using V.34 modems to 80 times slower using ISDN connections. Most mobile users use standard phone connections, while ISDN is becoming increasingly popular with telecommuters and remote offices.

Another remote access solution known in the art is remote control. Remote control allows users to execute applications from a remote location by taking over a host PC. The

only data transmitted across the phone line are the screen updates of the applications, and keystrokes made by the user. Remote control solutions trap graphics related function calls, transfer the data to the local machine and reproduce the output on the local machine. The application continues to run on the controlled PC, but is also displayed on the local machine which sends mouse and keyboard commands to the controlled PC.

Remote control products serve both the remote technical support market and the telecommuter and mobile user market. Since remote control technology allows users to take complete control over an entire PC, these products are well suited for remote technical support and helpdesk applications. Remote control is also today's solution of choice for telecommuters and mobile professionals who require remote access to their work. Remote control offers a high performance solution for users who need to remotely run high bandwidth applications on their own PCs.

As a solution for telecommuters and mobile users, remote control suffers from several disadvantages. From an organization's perspective remote control does not allow one server to support several remote users concurrently. Administering remote access and control configurations fast becomes a system administrator's nightmare. Single copies of applications cannot be installed on a single server to serve many remote users. This requires the need for a server per user, increasing drastically system complexity and maintenance costs. Thus, due to the one to one server to client ratio of this technology, it is unsuitable for large scale enterprise deployment, confining it to individual use and to the remote technical support market.

In spite of the availability of remote node and remote control solutions, none overcome the bandwidth limitation imposed by relatively slow phone lines. Even as remote node becomes ubiquitous, end users will increasingly demand solutions that enable them to remotely run corporate applications at LAN speed. What is needed is a solution combining remote node connectivity and one or more application servers. The solution should use remote node connectivity as the underlying communication layer, but enable remote users to run applications on a LAN based application server. The application server should be able to support multiple concurrent remote users.

Another solution known in the art is to provide an operating system based application server. This solution works similarly to remote control software, trapping graphics related function calls and reproducing them on the client. In addition, the

application server solution can support multiple concurrent users. This allows this solution to function as an application server, rather than simply a remote control server. however, since applications that run on the application server run entirely on the server, the host operating system must be modified to support multiple environments. This is required because typical operating systems, like Windows NT, for example, have only one desktop containing all graphic applications. Thus, several such desktops must be constructed, one for each concurrent user, and must be kept completely separate from one another. Using Windows NT as an example, items such as the clipboard, dynamic data exchange (DDE), and other inter application functions are not supported. To provide this functionality, the Windows NT operating system itself must be modified. The modified operating system, providing remote execution capabilities, would replace the original operating system.

The above solution has several disadvantages. From an organizational viewpoint, it is difficult to install because the entire operating system must be replaced and the existing system's functionality altered. Also, the software cannot be installed on existing operating system platforms, requiring a separate hardware platform. in addition, the problems inherent in a technology that simply executes the entire application on the server still remain. To the end user this means that most problems found in remote control solutions are still present in the modified operating system application sever solution. For example, no integration between local and remote applications is provided. Applications that use object linking and embedding (OLE), or dynamic data exchange (DDE), cannot function properly. Even cutting and pasting text between local and remote applications is not possible.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the present invention to provide a system for redirecting the user interface of an application, executing on a server computer, to a client computer located remotely from said server computer.

It is also an object of the present invention to provide a system wherein the user interface application programming interface function calls issued by an application are redirected from an operating system running on a server computer to an operating system running on a client computer.

Another object of the present invention is to provide a system wherein callback function calls issued by an operating system on a client computer are redirected from the client computer to the application running on the server computer.

Also an object of the present invention is to redirect the complete user interface of an application, running on a server computer, to a client computer wherein the redirection is transparent to a user.

Yet another object of the present invention is to provide a system that can support many concurrent users and is scaleable and suitable for enterprise wide deployment.

Another object of the present invention is to provide a system with improved performance when operating over slow communication lines.

In a first embodiment, the present invention extends an operating system, like Windows 95 or Windows NT, to allow applications to be used on one machine while actually executing on another machine. Most elements of the application execute on the server. The user interface elements of the application execute on the client. The result is that applications perform most operations, including input/output (I/O) intensive and CPU intensive operations, on the server but interact with the user on the local machine like any local application would. Multi-user capabilities are extended to support execution of applications, supporting multiple concurrent remote users. Utilizing the present invention, applications can execute on mixed architectures. For example, users of low cost Intel Windows 95 workstations can execute applications on powerful Alpha, PowerPC, or MIPS based Windows NT servers.

In a second embodiment, performance is further improved by providing a user interface skeleton or virtual user interface that locally generates the calls to the callback

functions of the application. Any callback function return values are passed to the client. This improves the performance of the application user interface redirector over slow connections such as modems over dial up lines.

Thus, there is provided, in accordance with a first embodiment of the present invention, a system for redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer which includes a redirector server receiving and translating into machine independent form the user interface application programming interface function call, a redirector client receiving and translating into machine dependent form the machine independent form of the function call, the machine dependent form of the function call able to be executed by the client computer, the redirector client receiving and translating into machine independent form a user interface callback call from the second operating system, the redirector server receiving and translating into machine dependent form the machine independent form of the callback call, the machine dependent form of the callback call able to be executed by the application on the server computer, the redirector client receiving and translating into machine independent form at least one return value from the second operating system in response to the function call, the redirector client translating into machine independent form any internal data modified as a result of the function call, the redirector server receiving and translating into machine dependent form the machine independent form of the at least one return value and the machine independent form of any modified internal data, the redirector server modifying internal data in the server computer in accordance with the machine independent form of modified internal data, the redirector server receiving and translating into machine independent form at least one return value from the application in response to the callback call, the redirector server translating into machine independent form any internal data modified as a result of the callback call, and the redirector client receiving and translating into machine dependent form the machine independent form of the at least one return value and the machine independent form of any modified internal data, the redirector client modifying internal data in the client computer in accordance with the machine independent form of modified internal data.

In accordance with a first embodiment of the present invention, the redirector server includes a function call interceptor receiving the function call issued by the application and

outputting machine dependent parameters in accordance with the function call, a translator translating the machine dependent parameters output by the function call interceptor into machine independent parameters, a callback translator translating machine independent parameters, associated with a callback function issued by the second operating system within the client computer, into machine dependent parameters, and a callback caller calling a callback function within the application utilizing the machine dependent parameters translated by the callback translator.

Also in accordance with a first embodiment of the present invention, the translator includes a function identifier translator translating the function identifier, output by the function call function interceptor, to a translation directive, a parameter splitter receiving the machine dependent parameters output by the function call interceptor and outputting a series of individual parameters and associated individual translation directives, an individual parameter translator coupled to the parameter splitter, the individual parameter translator generating a series of individual machine independent parameters, and a parameter collector coupled to the individual parameter translator, the parameter collector grouping the series of individual machine independent parameters into a machine independent function call.

In addition, in accordance with a first embodiment of the present invention the redirector client includes a translator translating machine independent parameters from the redirector server within the server computer into machine dependent parameters, a function caller calling the appropriate function call in the second operating system utilizing the machine dependent parameters translated by the translator, a callback interceptor receiving callback calls issued by the second operating system and outputting machine dependent parameters in accordance with the callback call, and a callback translator for translating machine dependent parameters from the callback interceptor into machine independent parameters.

In accordance with a first embodiment of the present invention, the translator includes a function identifier translator for translating the function identifier, output by the function call interceptor, to a translation directive, a parameter splitter for receiving the machine dependent parameters output by the function call interceptor and for outputting a series of individual parameters and associated individual translation directives, an individual parameter translator coupled to the parameter splitter, the individual parameter

translator for generating a series of individual machine independent parameters, and a parameter collector coupled to the individual parameter translator, the parameter collector for grouping the series of individual machine independent parameters into a machine independent function call.

In accordance with a first embodiment of the present invention, there is also provided a system for redirecting the user interface of an application, having at least one callback routine, from a server computer running a first operating system to a client computer running a second operating system, having at least one application programming interface routine, the system including trapping means coupled to the application, the trapping means for intercepting server user interface function calls issued by the application destined for the first operating system and directing said server user interface function calls to a redirector server, the redirector server, on the server computer, for receiving the server user interface function calls from the application, translating the server user interface function calls to machine independent user interface function calls and forwarding the machine independent user interface function calls to a redirector client, the redirector server for receiving machine independent user interface callback calls from the redirector client, translating the machine independent user interface callback calls into server user interface callback calls and forwarding the server user interface callback calls to the callback routine within the application, and the redirector client, on the client computer, for receiving the machine independent user interface function calls, translating the machine independent user interface function calls into client user interface function calls and forwarding the client user interface function calls to the second operating system, the redirector client for receiving the client user interface callback calls from the second operating system, translating the client user interface callback calls into machine independent user interface callback calls and forwarding the machine independent user interface callback calls to the redirector server.

In accordance with a first embodiment of the present invention, the trapping means performs a method of trapping the server user interface function calls, issued by the application, which includes the steps of loading an application loader, installing a first trap routine, loading the application using the application loader, initializing a first portion of the first operating system, executing the first trap routine which installs a second trap routine, initializing a second portion of the first operating system, loading the redirector

server using the application loader, and modifying the application so the server user interface function calls are made to the redirector server rather than the first operating system.

Also in accordance with a first embodiment of the present invention, there is provided a method of redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, which includes the steps of receiving and translating into machine independent form, on the server computer, the user interface application programming interface function call, receiving and translating into machine dependent form, on the client computer, the machine independent form of the function call, said machine dependent form of said function call able to be executed by said client computer, receiving and translating into machine independent form, on the client computer, a user interface callback call from the second operating system, receiving and translating into machine dependent form, on the server computer, the machine independent form of the callback call, the machine dependent form of the callback call able to be executed by the application on the server computer, receiving and translating into machine independent form, on the client computer, at least one return value from the second operating system in response to the function call, translating into machine independent form any internal data modified as a result of the function call, and server receiving and translating into machine dependent form, on the server computer, the machine independent form of the at least one return value and the machine independent form of any modified internal data, modifying internal data in the server computer in accordance with the machine independent form of modified internal data.

There is also provided in accordance with the present invention a system for redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, comprising a redirector server receiving and translating into machine independent form the user interface application programming interface function call, and a redirector client receiving and translating into machine dependent form the machine independent form of the function call, the machine dependent form of the function call able to be executed by the client computer.

Further, there is provided in accordance with the present invention, in a system for redirecting a user interface application programming interface (API) function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, a method for processing the API on the server computer, the method comprising the steps of trapping the API function call, transmitting the API function call to the second operating system on the client computer, determining on the server computer whether the API function call triggers a callback, calling the callback function of the application if the API does trigger a callback, translating any return parameters, and transmitting the return parameters to the second operating system on the client computer.

In addition, there is provided in accordance with the present invention, in a system for redirecting a user interface application programming interface (API) function call issued by an application, the application having at least one callback routine, the API function call destined for a first operating system running on a server computer, to a second operating system running on a client computer, a method for processing the API on the client computer, the method comprising the steps of receiving the API function call sent by the server computer, trapping a callback function issued by the second operating system without transmitting a corresponding message to the server computer, receiving any return parameters from the callback function executed on the server computer, updating the memory of the client computer with the return parameters, and returning any return values to the second operating system.

Also, there is provided in accordance with the present invention a system for redirecting the user interface application programming interface (API) function call issued by an application, the application having at least one callback routine, the API function call sent from a server computer running a first operating system to a client computer running a second operating system, the system comprising trapping means coupled to the application, the trapping means for intercepting API function calls issued by the application destined for the first operating system and directing the server user interface function calls to a translation means, the server translation means for receiving API function calls from the application, translating the API function calls to machine independent API function calls and forwarding the machine independent API function calls to a client translation means on the client computer, the client translation means for receiving the machine independent

API function calls, translating the machine independent API function calls into machine dependent API function calls and forwarding the API function calls to the second operating system, user interface skeleton means for tracking the states and attributes of substantially all user interface objects created on the client computer, a callback unit for synthesizing callback function calls to the callback routine in the application, the callback functions calls called ahead of their actual invocation by the API function on the client computer, the callback unit transmitting any return values from the callback routine to the client computer, and a callback interceptor for receiving and intercepting callback calls from the second operating system, the callback interceptor for providing any return values to the second operating system in accordance with the return values received from the callback unit on the server computer.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is herein described, by way of example only, with reference to the accompanying drawings, wherein:

Fig. 1 is a high level system block diagram illustrating a first embodiment of the present invention applied to a networking environment;

Fig. 2 is a high level flow diagram illustrating the startup procedure of the present invention;

Fig. 3 is a high level block diagram illustrating the client and server startup modules;

Fig. 4 is a high level functional block diagram illustrating the server portion of the first embodiment of the present invention;

Fig. 5 is a high level functional block diagram illustrating the client portion of the first embodiment of the present invention;

Fig. 6 is a high level functional block diagram illustrating the parameter translator;

Fig. 7 is a high level functional block diagram illustrating the parameter byte reversal translator;

Fig. 8 is a high level functional block diagram illustrating the machine dependent to machine independent pointer translator;

Fig. 9 is a high level functional block diagram illustrating the machine independent to machine dependent pointer translator;

Fig. 10 is a high level functional block diagram illustrating the message fields translator;

Fig. 11 is a high level system block diagram illustrating the second embodiment of the present invention having a user interface skeleton applied to a networking environment;

Fig. 12 is a high level functional block diagram illustrating the server portion of the second embodiment of the present invention;

Fig. 13 is a high level functional block diagram illustrating the client portion of the second embodiment of the present invention;

Fig. 14 is a high level flow diagram illustrating the server callback handler portion of the present invention; and

Fig 15 is a high level flow diagram illustrating the client callback handler portion of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

A high level block diagram illustrating a first embodiment of the present invention, generally referred to as redirector or system 10, as applied to a networking application, is shown in Figure 1. Redirector 10 comprises two portions, a server portion and a client portion. Located on the left hand side of the vertical dotted line in Figure 1 is a redirector server and on the right hand side, a redirector client. Both redirector server 16 and redirector client 20 can execute on personal computers, workstations and servers. The server computer runs an operating system (OS) 12 and the client computer runs an OS 22. OS 12 and OS 22 may or may not be similar operating systems, however, there must be a direct user interface application programming interface mapping between OS 12 and OS 22..

Server and client computers are coupled via a network 18. Network 18 may be any suitable network infrastructure such as a local area network (LAN), wide area network (WAN), FDDI, Ethernet, modem network connection, etc. using any suitable network protocol such as TCP/IP, IPX/SPX, NetBUEI, etc. Both redirector server 16 and redirector client 20 are coupled to network 18. On the server side, redirector server 16 interfaces with application 14. Application 14 also interfaces with OS 12 for network and other system related services. On the client side, redirector client 20 is coupled to network 18 and OS 22.

Application 14 is the application whose user interface is to be redirected. During the operation of system 10, non-user interface (non-UI) application programming interface (API) calls are sent to OS 12, as normal. An API function is an operating system provided function called by an application to request a service or request the operating system to perform some action. User interface API calls are trapped or intercepted and steered to redirector server 16. After being processed, the user interface API calls are sent to redirector client 20 over network 18. Based on the received user interface API calls, redirector client 20 generates a local API call appropriate for the local client computer and calls OS 22 with this local API. The window generated by application 14 is maintained by and presented to the user on the client computer. All user input data (i.e. mouse or other pointing device input, keyboard input, etc.) is received by the client computer and forwarded to the server. Similarly, all window updates and window drawing functions are

trapped on the server and forwarded to the client. Thus, a user interface shell of the application is created and maintained on the client computer while all other non-user interface functions of the application remain on the server computer. Return values from API function calls on the client computer are passed back to the server in a similar fashion.

System 10 thus permits a user to control an application, on a local computer, when the application is actually executing on a remote computer. The mechanism used to perform this function includes the trapping and redirection of all user interface API calls from the computer running the application (i.e. the server computer) to the user's remote computer (i.e. the client computer). Note that only user interface APIs are redirected, all other API calls execute on the server computer.

All trapped user interface API calls involve processing on both the server and client computers. API user interface calls must be processed because the environments on the server and client computers are typically not identical and thus do not permit the API function to execute correctly unmodified. Any reference to pointer or other data in the application's own address space is only meaningful on the server. The server and client computers may differ in terms of operating system, memory maps, pointer locations, API function call format, etc. Therefore, the original API issued by the application must be modified or translated before it can be properly called on the client computer.

Similarly, callback function calls are also redirected and translated for the server machine they are executed on. Typically, an application makes numerous API function calls to the OS in the course of its execution. Most operating systems, however, include a mechanism whereby the operating system itself may call a function included in the application. These calls made by the operating system are termed callback function calls. Similar to the API function call translation performed by system 10, translation of the operating system callbacks must also be performed. In addition, most operating systems require callback functions to be registered by the application before they can be called. Within system 10, the redirector client 20 registers callback stubs on the client computer. In practice, OS 22 calls one of the previously created stubs. Subsequently, the corresponding callback function is executed on the server and the return value returned to the client. The translation of both user interface API function calls and callbacks will be described in more detail below.

Referring to Figure 1, translated user interface API function calls are sent from redirector server 16 to the client. Redirector client 20 further translates the API function call and makes the local call to OS 22. Return values and any modified parameters are sent back to redirector server 16. Callbacks issued by OS 22 are translated by redirector client 20 and forwarded to redirector server 16. Redirector server 16 further translates the callback function and performs the actual call to application 14. Return values are translated and forwarded to OS 22 on the client computer. Return values, from both API and callback functions may include one or more modified parameters that also require translation.

Before user interface API function calls, generated by application 14, can be redirected to the client computer, an API function call trapping mechanism is installed. A method of installing a trapping mechanism is illustrated in Figure 2. In general, redirector server 16 library modules are inserted into the application's protected address space and initialized before any of the application's library modules are initialized but after essential operating system library modules are initialized, because the operating system library modules are needed by redirector server 16 library modules. In Figure 2, solid boxes indicate normal procedure normally performed by the operating system, whereas dotted boxes indicate additional steps required to install the trapping mechanism. The following trapping mechanism installation procedure, described below, will work on a computer running an operating system such as Windows NT, manufactured by Microsoft Corporation, Redmond, Washington. The client portion of the present invention will also work on other operating systems. One skilled in the art can readily adapt the embodiment disclosed herein to other operating systems.

To initialize system 10, a user first launches redirector client 20 (Figure 1) on the client computer. The user selects the application whose user interface is to be redirected. Redirector client 20 causes redirector server 16 and chosen application 14 to be launched and initialized in the following manner. An application loader is first loaded (i.e. from a hard disk or other long term memory storage) into memory (step 24). Next, a first trap routine is installed and later executed after the operating system kernel is initialized (step 26). The CreateProcess function is used to create a process (i.e. application 14) in suspended mode. The CreateRemoteThread function is used to create a suspended thread in the remote process. This thread is used to find stack space in the remote process. The

stack space is allocated to a first trap routine. A jump instruction to the first trap routine in the CsrClientConnectToServer routine in NTDLL is inserted. The process is then unsuspended. Trapping is installed at this stage because otherwise security within the kernel would not allow the trapping mechanism to be put into place. In step 28, the application modules are loaded into memory and the operating system core is initialized (NTDLL in the Windows NT operating system). In step 30, kernel non-user interface APIs are initialized. When the CsrClientConnectToServer function is reached, after the application is loaded into memory but before its DLLs or library modules are initialized, the inserted jump instruction causes control to be diverted to the first trap routine.

The first trap routine first restores the code in the CsrClientConnectToServer function. It then searches through the application process' DLLs (e.g., NTDLL, KERNEL32, USER32 and GDI32). After finding the first application DLL, the location of its initialization code is determined and a jump instruction to a second trap routine is placed in memory (step 32). Next, the essential Windows DLLs (i.e. user interface API functions: USER and GDI APIs) are initialized (step 34). Control is diverted to the second trap routine when execution reaches the modified DLL initialization code, after the Windows DLLs are initialized.

Referring to Figures 2 and 3, the second trap routine first restores the initialization code to its original state. It then calls the LoadLibrary function to load redirector server 16 library modules or DLLs into memory (step 36). At this point redirector server 16 is loaded into memory and its initialization module 48 (Figure 3) is executed (step 38). During initialization of redirector server 16, application 14 is modified to call redirector server 16 rather than OS 12 when making user interface API function calls. Initialization module 48 functions to locate the Image Activation Tables for each DLL in the application process. These operating system tables are searched for all user interface APIs that are to be trapped. The tables are then modified to point to short redirector trapping stubs 46, which contain a call to an API call interceptor 50.

Shown in Figure 3 is application 14 and redirector server 16. The loading of redirector server 14 library modules or DLLs (step 36, Figure 2), comprises the loading of an initialization module 48, API call interceptor 50 and redirector modules 52. As previously discussed, during initialization, redirector trapping stubs 46 are installed for all user interface function calls used by application 14. This causes all user interface API

function calls issued by application 14 to be redirected to API call interceptor 50. Non-user interface API function calls completely bypass the trapping mechanism and call OS 12 directly. Redirector modules 52 are used during the redirection and translation process, discussed below.

The operation of system 10 will now be described in more detail by illustrating the path an API function call traverses going from the server to the client and the path back to the server taken by the return values. A more detailed block diagram of the redirector server 16 is illustrated in Figure 4. Application 14 issues both user interface and non-user interface API function calls. The trapping mechanism, discussed above, causes all user interface API function calls to be directed to API call interceptor 50. Non-user interface API function calls continue to be handled by the operating system. For each API function call, API call interceptor 50 assigns a corresponding API translation directive. In addition, API call interceptor 50 takes the API function call parameters from the processor stack and generates a machine dependent parameter stream. The translation directive is determined according to which API function was called and according to the type of parameters that accompany the API function call. The parameters of the API call are parsed and classified into appropriate translation directives. API call interceptor 50 maintains a lookup table containing entries for all possible API function calls that may be redirected to it. For each API function entry there is a corresponding translation directive. The lookup table may contain on the order of hundreds of entries, depending on the number of API functions redirected. Each translation directive indicates how the machine dependent parameters will be processed by a server API translation unit 86. The translation directive itself may be a unique token or keyword defining the type of parameter translation to be performed.

Translation directives are needed because the API function call, as it is issued on the server computer, cannot be executed directly on the client computer. For example, pointers to memory locations on the server have no relation to and do not correspond to the same memory location on the client. Parameters passed by reference (i.e. memory pointers) do not exist on the client computer. Therefore, for pointer parameters, it is necessary to send the data the pointer points to rather than the pointer itself. Any parameter that references data on the server rather than the data itself, must be translated or expanded to encompass all the data. Any reference to system data must also be expanded

to include the data itself, not simply the pointer reference to it. The translation process is performed by server API translation unit 86.

Server API translation unit 86 translates or expands machine dependent parameters into machine independent parameters. All pointers to local data structures are eliminated. Any machine specific data (i.e. arrays, pointers, system registers, etc.) is translated into a machine independent form. Any operating system data to execute the API function on the client side is fetched from memory. The translation is performed according to the API translation directive received from API call interceptor 50. Translation unit 86 tracks each API call that it received from API call interceptor 50. This is to ensure that it can match received return values to the corresponding API function call and perform a reverse translation. Translation unit 86 outputs an API ID in addition to machine independent parameters. The API ID is a token, keyword or unique identification value indicating the API function called. In addition, the translation unit 86 may output an internal command. The internal command is issued by the server portion and not directly by the application. Internal commands are received and interpreted by the client side portion. They are used to query the state of the client system or the user interface object executing on it, to group several Windows APIs for performance reasons or to 'envelope' a Windows API with additional data it may require in order to execute correctly on the client. The format and encoding of the internal command is similar to that of Windows APIs.

The output of translation unit 86 is sent to a transport unit/protocol parser 80. Transport unit/protocol parser 80 processes the data received and sends it to OS network support 82 which places the data onto network 18. Referring to Figure 5, the data is received by OS network support 108 on the client computer. The OS passes the data to client transport unit/protocol parser 106. The machine independent parameters and API ID or internal command are received by a client API translation unit 104. Client translation unit 104 performs the opposite function of server translation unit 86 (Figure 4). Machine independent parameters are translated into machine dependent parameters. The parameters output by translation unit 104 reference data located on the client computer. The machine independent data sent from the server computer is placed into memory on the client computer. Memory blocks are allocated on the client computer to hold this data. For data originally referenced by pointers the machine dependent data output by translation unit 104 is also a pointer. However, the pointer output by translation unit 104 points to an address

in memory on the client computer rather than on the server computer. According to the API ID, translation unit 104 performs the reverse translation process of generating a parameter stream suitable for a local API call on the client computer. For example, data structures received from the server computer are replaced with a corresponding pointer value, after the data structure is placed in memory on the client computer. In addition the translation unit 104 performs the internal command transmitted from the server portion.

The machine dependent parameters and the API ID are output by translation unit 104 to an API caller 100. API caller 100 performs the actual call to OS 22. From the API ID, API caller 100 determines the proper address to call on the client computer. On the client computer, a lookup table is generated which contains entries for all possible API IDs and their corresponding actual addresses on the client computer. This lookup table is generated once during the launch of redirector client 20.

API caller 100 makes the actual API call to the corresponding API routine 96 within OS 22. As discussed previously, a shell process representing the user interface of the application is maintained on the client computer. The API function calls that issue on the client computer modify the window of the application on the screen. Thus, to the user, it appears that the application itself is executing on the client computer. In actuality, however, only the user interface portion of the application is executing on the client computer. The non-user interface portion of the application executes on the server computer.

Any return values generated by API routine 96 are passed back to API caller 100, the calling module. API caller 100 passes the return value to client API translation unit 104. Any return values that are machine dependent (i.e. local memory pointers, client OS 22 reference data or memory locations, modified data structures in client computer memory, etc.) must be translated to an independent form. Thus, client API translation unit 104 performs a similar function on the return data, as server API translation unit 86 (Figure 4) performs on API call parameters. The return value and modified parameters are passed to transport unit 106 and subsequently to OS network support 108 and network 18. The data is received by server API translation unit (Figure 4) after handling by OS network support 82 and transport unit 80. Translation unit 86 matches the data, which includes both return value and modified parameters, with the called API function to determine how to translate the modified parameters back into machine dependent form for the server

computer. It synchronizes the existing server computer memory data with the modified parameters it received from the client computer. For example, if the API call sent to the client computer was to allocate a memory block, the return value would be an address pointing to the created memory block. Since the server has no access to memory on the client, it must explicitly allocate a memory block of the same size on the server, copy the contents and substitute the pointer to the server memory block. The modified parameters received from the client computer contain sufficient data to replicate what occurred during the API call on the client computer, on the server computer. Once the modified parameters are reverse translated for the server computer, the return value is passed to the API call interceptor 50 and finally back to application 14 that originally made the API function call.

Referring to Figures 4 and 5, the callback process will now be described in more detail. As previously discussed, callback calls are made by the operating system to invoke a callback routine supplied by the application. In Figure 4, callback routine 54 provided by application 14 is called by OS 22. Before callbacks can be made, they must first be registered. Typically, application 14 registers its callback functions upon application startup, however. they may be registered at any time. The registration APIs are trapped, intercepted and eventually are received by Client API translation unit 104. Client API translation unit 104 replaces the actual callback address of application 14 with a phantom stub address on the client computer. The callback API call is then sent to OS 22. The callback stubs are installed so as to trap and intercept all callback calls to a callback interceptor 98. Callback interceptor 98 performs a similar function as API call interceptor 50. Callback interceptor 98 traps the callback call, parses the call contents and outputs a machine dependent parameter stream and a callback translation directive to a client callback translation unit 102. Client callback translation unit 102 performs a similar translation process on the machine dependent parameters as server API translation unit 86. Translation unit 102 translates the machine dependent parameters to machine independent parameters. It also outputs a callback ID in place of the actual callback call. The data output is then transferred over network 18 to a server callback translation unit 88. Callback translation unit 88 functions similarly as client API translation unit 104. Machine independent parameters are translated to machine dependent parameters. The machine dependent parameters and the callback ID are output to a callback caller 90. Callback caller 90 functions to take the machine dependent parameters and the callback ID and

generate the actual local callback function call. Callback caller 90 calls callback routine 54 within application 14.

Return values and any modified parameters are passed through to callback translation unit 88 by callback caller 90. Callback translation unit 88 processes the return values and any modified parameters similarly to client API translation unit 104. Any machine dependencies within the return values are removed. The modified parameters are transmitted to client callback translation unit 102 where they are processed into return values and modified parameters referenced to the client computer. The output return value in addition to any modified parameters are passed to callback caller 98 which passes the return value to OS 22, the original caller.

Illustrated in Figure 6 is a parameter translator 140. Parameter translator 140 is representative of the operation of both translation units 86, 88 on the server computer and translation units 104, 102 on the client computer. The operation of parameter translator 140 depends on how it is invoked. In addition, individual parameter translator 146 encapsulates the parameter translators illustrated in Figures 7-10. Translator 140 comprises a translator 142 coupled to a parameter splitter 144. Individual parameter translator 146 is coupled to parameter splitters 144, 152 and parameter collectors 148, 150. There are two different operating modes for parameter translator 140. The first mode represents server API translation unit 86 and client call translation unit 102. The other mode represents server callback translation unit 88 and client API translation unit 104. The operation of parameter translator 140 will now be described for each of the two possible modes of operation.

In the first mode of operation, the API or callback translation directive is input to translator 142 which inputs the directive unchanged to parameter splitter 144. Translator 142 looks up, in its look up table, the appropriate API or callback ID from the translation directive received. Parameter splitter 144 generates an ordered collection of individual or single parameters. Each individual parameter has associated with it a parameter translation directive. Each individual or single parameter and its associated translation directive are passed to individual parameter translator 146. Translator 146, in the first mode of operation, translates each single dependent parameter to single independent translated parameters. A more detailed description of translator 146 is discussed below. A parameter collector 148 collects and stores all the single translated parameters output from translator

146 and outputs a stream of translated parameters after all the single parameters have been translated.

Return values and modified parameters are input to parameter splitter 152 which outputs each individual or single parameter to translator 146. All individual translated parameters are collected by parameter collector 150 which outputs a translated return value.

The procedure of the second mode of operation is similar to that of the first. Machine independent parameters are input to parameter splitter 144. The API or callback ID is input to translator 142 which outputs the appropriate directive based on the ID. The ID is input to splitter 144 which parses the input parameters and outputs a stream of single parameters with their associated translation directives. The parameters undergo a translation from independent to dependent form. The single translated parameters are collected and output by collector 148. Return values are input to splitter 152. The single translated return values are collected and output by collector 150.

Illustrated in Figure 7 is a simple parameter translator. The function of the simple parameter translator is to ensure that the byte order of all parameters are uniform before they are transmitted over the network. The translator utilizes the parameter translation directive to determine if the byte order of a particular parameter should be reversed. A byte reversing means 154 reverses the byte order of an untranslated parameter to form a translated parameter. Another byte reversing means 156 reverses the byte order of an untranslated return parameter to a translated return parameter.

Illustrated in Figures 8 and 9 are pointer translators for translating machine independent parameters to and from machine dependent parameters. As discussed previously, Figures 8 through 10 illustrate different embodiments of individual parameter translator 146 (Figure 6). As mentioned earlier, parameters are translated before they are sent to another computer because the actual operating system environment on each computer is different. Even though each computer (i.e. both client and server) may have identical operating systems, the memory map on each will differ. The locations of variables stored on one computer will not match those, or even exist, on another computer.

Referring to Figures 6 and 8, the machine dependent to machine independent translator will be discussed first. A parameter parser 160 for receiving machine dependent parameters is coupled to a length determiner 162. Parser 160 extracts the pointer address

and related data from the input parameter using the parameter translation directive. The directive serves as a guide as to how to translate the parameter. Length determiner 162 determines the length of the data pointed to by the parameter pointer. The expanded parameter data is input to nested parameter translator 140. If the pointer data output by length determiner 162 is not a data pointer, then the pointer data passes through the simple parameter translator illustrated in Figure 7 that reverses, if necessary, the byte order of the parameter. If, however, the pointer data is another pointer itself, then the pointer data is passed through the parameter translator illustrated in Figure 6 again. Thus, nested parameter translator 140 is invoked recursively. A hierarchy of nested pointer data is created with another level being added to the hierarchy for each pointer within the pointer data. Each pointer in the hierarchy is expanded until the data does not contain a pointer. For each nested data pointer another instance of individual parameter translator 146 is invoked. In turn, each set of data associated with each pointer in the hierarchy is expanded and its corresponding parameters collected by the particular instance of parameter collector 148. The output of nested parameter translator 140 is a stream of machine independent translated parameters which include the length of the data, the data represented by the pointer and other related data, such as various operating system data. The output translated parameter data is collected by parameter collector 148 and output to transport unit 80, 106, for server and client respectively.

Return parameters are treated in an opposite fashion. Machine independent return parameters are input to parser 168. The input data includes the data length, previously expanded pointer and other related data. This data is input to nested parameter translator 140. If needed, the simple parameter translator performs byte reversal. Utilizing the translation directive, nested parameter translator 140 determines the pointer locations and the appropriate data to insert in the data memory pointed to by the pointer. Output assembler 166 writes the pointer address to the pointer location and the data contents to the data memory block location.

An embodiment of individual parameter translator 146 for translating machine independent parameters to machine dependent parameters is illustrated in Figure 9. Reference will also be made to Figure 6. Operation of the pointer translator illustrated in Figure 9 is analogous to that of Figure 8 except that machine independent parameters are translated into machine dependent parameters. Machine dependent parameters are input to

parser 170 which extracts the length of the data, the pointer and other related data associated with the parameter. The pointer data is then input to nested parameter translator 140 which, while utilizing the parameter translation directive, recursively determines (if necessary) the pointer locations and the appropriate data to insert in the data memory pointed to by the pointer. The translated pointer data is input to output assembler 172 which writes the local machine dependent parameters to the appropriate memory locations. Output assembler 172 functions similarly to output assembler 166 (Figure 8). The independent data received is localized, placed in the appropriate memory locations and the corresponding pointer is assigned the memory address of the data.

Return parameters are received by parser 178 which reads the pointer and its associated data. Length determiner 176 figures the length of the pointer data and inputs it to nested parameter translator 140 which operates recursively as described above. Translated pointer data is input to output assembler 174 which writes the pointer, length, pointer and other related data to parameter collector 150. As described previously, output collector 150 receives each individual translated parameter, groups them together appropriately and outputs a translated return value.

Illustrated in Figure 10 is a message field translator for converting machine dependent messages to and from machine independent messages. A message API is an operating mechanism used to inform an application of asynchronous events. The message and its parameters are input to a message extractor 180 which extracts the fields of the message from the input. The message fields are the window identifier, message ID and the message specific parameters. The message fields are input to message fields translator 184. Message fields translator functions similarly to nested translator 140 (Figures 8 and 9) except that translator 184 operates on messages which requires an inspection of the contents of the message in order to determine how to translate it. This is in contrast to translating API function calls in that the API function call itself is self identifying.

The message ID output by message extractor 180 is input to directive lookup table (LUT) 182. Directive LUT determines a message translation directive from the message ID from a lookup table it maintains. The message translation directive is input to message fields translator 184. In the dependent to independent mode of operation, pointer parameters output by message extractor 180 are input to fields translator 184 which outputs

the data corresponding to the input pointer. The translated output message fields are then collected by parameter collector 148 (Figure 6).

Return parameters are input to message extractor 186 which function similarly to message extractor 180. The window identifier, message ID and other message specific parameters are extracted from the input return parameters. The resulting message fields are input to message fields translator 184 which processes the message fields to output translated message fields.

A second embodiment of the present invention improves the performance of the redirector 10 when the server and client portions communicate over relatively slow communication lines, e.g., dial up modem connections. A bottleneck is created due to the dependency on long latency periods typical of such slow connections. The second embodiment described herein minimizes the bottleneck to performance caused by such slow communication lines. The second embodiment comprises a user interface (UI) skeleton or virtual UI framework that is maintained by the server portion of the redirector. The UI skeleton tracks all UI objects created on the client side, i.e., the respective states and attributes of each UI object created on the client are tracked. In addition, calls to callback functions are synthesized by the server side, based on the APIs called by the application and the data stored in the UI skeleton and are called ahead of their actual invocation by API code running on the client. Further, the communication between the server and the client portion includes, in addition to the APIs, internal commands and callback return parameters.

A high level system block diagram illustrating the second embodiment of the present invention having a user interface skeleton applied to a networking environment is shown in Figure 11. Translated user interface API function calls, internal commands and callback return values are sent from redirector server 16 to the client. Redirector client 20 further translates the API function call or the internal command and makes the local call to OS 22 or executes the internal command, respectively. Return values and any modified parameters are sent back to redirector server 16. Unlike the case of the redirector of Figure 1, callbacks issued by OS 22 are not translated by redirector client 20 and forwarded to redirector server 16. In this case, the redirector server 16 translates the callback function and performs the actual call to application 14 but not, however, in response to a call from

the client. The redirector server synthesizes the call based on APIs and data maintained by the UI skeleton.

A high level functional block diagram illustrating the server portion of the second embodiment of the present invention is shown in Figure 12. The operation of the redirector server 16 is similar to that of Figure 4 with the exception of the server API translation unit 86 and server callback translation unit 88. The server callback translation unit is eliminated and the server API translation unit is replaced by a user interface skeleton 190, i.e., a virtual UI, and a callback unit 192. Callbacks originating on the client are not translated and sent to the server redirector.

The UI skeleton 190 is operative to emulate the user interface objects that exist on the client side. For each user interface object, e.g., windows, dialogs, controls, menus, etc., created on the client, a corresponding skeleton object is constructed on the server side. The skeleton keeps track of the actual state and attributes of the object. Using the information stored in the skeleton, the server redirector can synthesize the return values and modified variables for API calls issued by the application without having to wait for return values from the client side. This eliminates the performance bottleneck due to the latency of the communication line when waiting for return values. The application now receives the return values it requested directly from the server typically before the API even finishes executing on the client.

Further, the UI skeleton 190/callback unit 192 may output an internal command in addition to the API to the transport unit/protocol parser 80. The internal command is issued by the server portion and not directly by the application. Internal commands are received and interpreted by the client side portion. They are used to query the state of the client system or the user interface object executing on it, to group several Windows APIs for performance reasons or to 'envelope' a Windows API with additional data it may require in order to execute correctly on the client. The format and encoding of the internal command is similar to that of Windows APIs.

The callback unit 192 in combination with the UI skeleton 190 functions to generate the machine dependent parameters and the callback ID for the callback caller 90 in accordance with the particular API called by the application 14. The return value generated by the callback routine is returned to the callback caller 90. The callback caller

90 processes this data and outputs a return value and any modified parameters to the callback unit 192.

In similar fashion, the redirector client 20 is also modified so that callbacks are not translated and sent to the redirector server. A high level functional block diagram illustrating the client portion of the second embodiment of the present invention is shown in Figure 13. The same named elements of Figure 13 operate in a similar fashion with those of Figure 5 except for the callback interceptor 230 and the client callback translation unit 232.

The callback interceptor 230 now only processes return values and any modified parameters transmitted from the client callback translation unit 232. Based on the data input to the callback interceptor 230, it forms the return value sent to the OS 22. The callback interceptor 230 does not send machine dependent parameters and a callback translation directive to the client callback translation unit 232.

Similarly, the client callback translation unit 232 only processes return values and any modified parameters transmitted sent from the redirector server and sends translated return data to the client interceptor 230. The client callback translation unit 232 does not send machine independent parameters and a callback ID to the redirector server.

The server redirector must handle callbacks properly if the UI skeleton is to be effective with any API that may trigger a callback by the OS on the client side. If this is not so, the application may have to stop executing, wait for the client to issue the callback, execute the application code for the callback and send any return parameters back to the client. To eliminate this bottleneck and to ensure that the correct order of execution is preserved on both the server and the client, the following two methods are utilized to handle callbacks.

The first method, a high level flow diagram illustrating the server callback handler portion of the present invention, is shown in Figure 14. First, the API call is trapped, packed and sent to the client (step 194). This is performed by the redirector server 16 of Figure 12 and the redirector client 20 of Figure 13. Next, the UI skeleton 190 determines whether the API will trigger or generate a callback (step 196). If so, the callback unit 192 issues a call to the callback function of the application program (step 198). The callback ID and related parameters are sent to the callback caller 90 (Figure 12). Next, the return parameters are received from the callback routine 54 through the callback caller 90. They

are then translated and packed into one or more internal commands (step 200). These internal commands are then sent to the redirector client (step 202).

The second method, a high level flow diagram illustrating the client callback handler portion of the present invention, is shown in Figure 15. On the client side, the API is received by the transport unit/protocol parser 106, unpacked, translated by the client API translation unit 104 and called by the API caller (step 210). If the API has a callback associated with it, the OS 22 issues a callback call to the callback interceptor 230 (step 212). The callback is trapped by the callback interceptor but nothing is sent to the redirector server (step 214). If the API triggered a callback on the server (step 216) any callback return values that the callback routine generates, sent to the client via one or more internal commands, are received and unpacked (step 218). The computer's memory is then appropriately updated (step 220) and any return values are returned to the OS 22 (step 222).

It is important to point out that in the methods of Figures 14, 15, the callback is executed on the server before it is actually called by the client OS. Thus, the client and the server execute asynchronously. This enables the application to continue executing without having to wait for the client OS to issue the callback, while at the same time preserving the correct order of execution at both the server and the client.

The protocol between the server redirector and the client redirector comprises a stream of records wherein each record can have one of three types. The first type is a Windows API, wherein the server sends to the client each API as an API ID, following by the translated parameters of the API. If requested, the client responds with a record comprising return parameters for the particular API. Return parameters include any return values and variables that were passed by reference to and changed by the API.

The second type is an internal command which is issued by the redirector server and not directly by the application. The purpose and use of internal commands have been described in detail above.

The third type are callback return parameters which include any return values of the callback function and the content of variables passed by reference to and changed by the callback function. The return parameters are translated, packed, marked as callback parameters and sent from the server to the client.

While the invention has been described with respect to a limited number of embodiments, it will be appreciated that many variations, modifications and other applications of the invention may be made.

CLAIMS

1. A system for redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, comprising:

a redirector server receiving and translating into machine independent form said user interface application programming interface function call;

a redirector client receiving and translating into machine dependent form said machine independent form of said function call, said machine dependent form of said function call able to be executed by said client computer;

said redirector client receiving and translating into machine independent form a user interface callback call from said second operating system;

said redirector server receiving and translating into machine dependent form said machine independent form of said callback call, said machine dependent form of said callback call able to be executed by said application on said server computer;

said redirector client receiving and translating into machine independent form at least one return value from said second operating system in response to said function call, said redirector client translating into machine independent form any internal data modified as a result of said function call;

said redirector server receiving and translating into machine dependent form said machine independent form of said at least one return value and said machine independent form of any modified internal data, said redirector server modifying internal data in said server computer in accordance with said machine independent form of modified internal data;

said redirector server receiving and translating into machine independent form at least one return value from said application in response to said callback call, said redirector server translating into machine independent form any internal data modified as a result of said callback call; and

said redirector client receiving and translating into machine dependent form said machine independent form of said at least one return value and said machine

independent form of any modified internal data, said redirector client modifying internal data in said client computer in accordance with said machine independent form of modified internal data.

2. The system as claimed in claim 1, wherein said redirector server comprises:

a function call interceptor receiving said function call issued by said application and outputting machine dependent parameters in accordance with said function call;

a translator translating said machine dependent parameters output by said function call interceptor into machine independent parameters;

a callback translator translating machine independent parameters, associated with a callback function issued by said second operating system within said client computer, into machine dependent parameters; and

a callback caller calling a callback function within said application utilizing said machine dependent parameters translated by said callback translator.

3. The system as claimed in claim 2, wherein said translator comprises:

a function identifier translator translating said function identifier, output by said function call interceptor, to a translation directive;

a parameter splitter receiving said machine dependent parameters output by said function call interceptor and outputting a series of individual parameters and associated individual translation directives;

an individual parameter translator coupled to said parameter splitter, said individual parameter translator generating a series of individual machine independent parameters; and

a parameter collector coupled to said individual parameter translator, said parameter collector grouping said series of individual machine independent parameters into a machine independent function call.

4. The system as claimed in claim 1, wherein said redirector client comprises:

a translator translating machine independent parameters from said redirector server within said server computer into machine dependent parameters;

a function caller calling said application utilizing said machine dependent parameters translated by said translator;

a callback interceptor receiving callback calls issued by said second operating system and outputting machine dependent parameters in accordance with said callback call; and

a callback translator for translating machine dependent parameters from said callback interceptor into machine independent parameters.

5. The system as claimed in claim 4, wherein said translator comprises:

a function identifier translator for translating said function identifier, output by said function call interceptor, to a translation directive;

a parameter splitter for receiving said machine dependent parameters output by said function call interceptor and for outputting a series of individual parameters and associated individual translation directives;

an individual parameter translator coupled to said parameter splitter, said individual parameter translator for generating a series of individual machine independent parameters; and

a parameter collector coupled to said individual parameter translator, said parameter collector for grouping said series of individual machine independent parameters into a machine independent function call.

6. A system for redirecting the user interface application programming interface (API) function call issued by an application, said application having at least one callback routine, said API function call sent from a server computer running a first operating system to a client computer running a second operating system, said system comprising:

trapping means coupled to said application, said trapping means for intercepting API function calls issued by said application destined for said first operating system and directing said API function calls to a server redirection means;

said server redirection means, on said server computer, for receiving said API function calls from said application, translating said API function calls to machine independent user interface function calls and forwarding said machine independent API function calls to a client redirection means, said

server redirection means for receiving callback calls from said client redirection means, translating said callback calls into machine dependent callback calls and forwarding said machine dependent callback calls to said callback routine within said application; and

said client redirection means, on said client computer, for receiving said machine independent API function calls, translating said machine independent API function calls into machine dependent API function calls and forwarding said machine dependent API function calls to said second operating system, said client redirection means for receiving said callback calls from said second operating system, translating said callback calls into machine independent callback calls and forwarding said machine independent callback calls to said server redirection means.

7. The system as claimed in claim 6, wherein said trapping means performs a method of trapping said API function calls, issued by said application, comprising the steps of:

loading an application loader;

installing a first trap routine;

loading said application using said application loader;

initializing a first portion of said first operating system;

executing said first trap routine which installs a second trap routine;

initializing a second portion of said first operating system;

loading said redirector server using said application loader; and

modifying said application so said API function calls are made to said redirector server rather than said first operating system.

8. The system as claimed in claim 6, wherein said server redirection means comprises:

a call interceptor for receiving said API function calls from said application and outputting machine dependent parameters and a translation directive, said call interceptor for receiving at least one return value and outputting said at least one return value to said application;

a translator for receiving said machine dependent parameters and said translation directive, said translator for translating said server machine dependent

parameters into machine independent parameters utilizing said translation directive, said translator outputting said machine independent parameters and a function identifier representing said API function call, said translator for writing machine independent modified parameters returned from said client computer into server computer memory;

a callback translator for receiving said machine independent parameters and a callback identifier from said client computer, said callback translator for translating said machine independent parameters into machine dependent parameters utilizing said callback identifier, said callback translator outputting said machine dependent parameters and said callback identifier, said callback translator for translating received return values into independent modified parameters; and

a callback caller for receiving said machine dependent parameters and said callback identifier from said callback translator and calling said callback routine within said application, said callback caller for receiving at least one return value and outputting said at least one return value to said callback translator.

9. The system as claimed in claim 6, wherein said client redirection means comprises:

a callback interceptor for receiving said callback calls from said second operating system and outputting client machine dependent parameters and a translation directive, said callback interceptor for receiving at least one return value and outputting said at least one return value to said second operating system;

a callback translator for receiving said client machine dependent parameters and said translation directive, said callback translator for translating said client machine dependent parameters into machine independent parameters utilizing said translation directive, said callback translator outputting said machine independent parameters and a callback identifier representing said callback call, said callback translator for writing machine independent modified parameters returned from said server computer into client computer memory;

a translator for receiving said machine independent parameters and a function identifier from said server computer, said translator for translating said machine independent parameters into machine dependent parameters utilizing said function identifier, said translator outputting said machine independent parameters and said function identifier, said translator for translating received return values into independent modified parameters; and

a function caller for receiving said machine dependent parameters and said function identifier from said translator and calling said API routine within said second operating system, said function caller for receiving at least one return value and outputting said at least one return value to said translator.

10. The system as claimed in claim 6, wherein said server redirection means comprises:

a call interceptor for receiving said API function calls from said application and outputting server machine dependent parameters and a translation directive, said call interceptor for receiving at least one return value and outputting said at least one return value to said application;

server translation means for receiving API function calls from said application, translating said API function calls to machine independent API function calls and forwarding said machine independent API function calls to a client translation means on the client computer;

user interface skeleton means for tracking the states and attributes of substantially all user interface objects created on the client computer; and

a callback unit for synthesizing callback function calls to the callback routine in the application, said callback functions calls called ahead of their actual invocation by the API function on the client computer, said callback unit transmitting any return values from the callback routine to the client computer.

11. The system as claimed in claim 6, wherein said client redirection means comprises:

client translation means for receiving said machine independent API function calls, translating said machine independent API function calls into machine

dependent API function calls and forwarding said API function calls to said second operating system; and

a callback interceptor for receiving and intercepting callback calls from said second operating system, said callback interceptor for providing any return values to said second operating system in accordance with the return values received from the server computer.

12. A method of redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, the method comprising the steps of:

receiving and translating into machine independent form, on said server computer, said user interface application programming interface function call;

receiving and translating into machine dependent form, on said client computer, said machine independent form of said function call, said machine dependent form of said function call able to be executed by said client computer;

receiving and translating into machine independent form, on said client computer, a user interface callback call from said second operating system;

receiving and translating into machine dependent form, on said server computer, said machine independent form of said callback call, said machine dependent form of said callback call able to be executed by said application on said server computer;

receiving and translating into machine independent form, on said client computer, at least one return value from said second operating system in response to said function call, translating into machine independent form any internal data modified as a result of said function call; and

server receiving and translating into machine dependent form, on said server computer, said machine independent form of said at least one return value and said machine independent form of any modified internal data, modifying internal data in said server computer in accordance with said machine independent form of modified internal data.

13. A system for redirecting a user interface application programming interface function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, comprising:

a redirector server receiving and translating into machine independent form said user interface application programming interface function call; and

a redirector client receiving and translating into machine dependent form said machine independent form of said function call, said machine dependent form of said function call able to be executed by said client computer.

14. In a system for redirecting a user interface application programming interface (API) function call issued by an application and destined for a first operating system running on a server computer, to a second operating system running on a client computer, a method for processing said API on said server computer, said method comprising the steps of:

trapping the API function call;

transmitting the API function call to the second operating system on the client computer;

determining on the server computer whether the API function call triggers a callback;

calling the callback function of the application if the API does trigger a callback;

translating any return parameters; and

transmitting the return parameters to the second operating system on the client computer.

15. In a system for redirecting a user interface application programming interface (API) function call issued by an application, said application having at least one callback routine, said API function call destined for a first operating system running on a server computer, to a second operating system running on a client computer, a method for processing said API on said client computer, said method comprising the steps of:

receiving the API function call sent by the server computer;

trapping a callback function issued by the second operating system without transmitting a corresponding message to the server computer;

receiving any return parameters from the callback function executed on the server computer;

updating the memory of the client computer with the return parameters; and

returning any return values to the second operating system.

16. A system for redirecting the user interface application programming interface (API) function call issued by an application, said application having at least one callback routine, said API function call sent from a server computer running a first operating system to a client computer running a second operating system, said system comprising:

trapping means coupled to said application, said trapping means for intercepting API function calls issued by said application destined for said first operating system and directing said server user interface function calls to a translation means;

said server translation means for receiving API function calls from said application, translating said API function calls to machine independent API function calls and forwarding said machine independent API function calls to a client translation means on the client computer;

said client translation means for receiving said machine independent API function calls, translating said machine independent API function calls into machine dependent API function calls and forwarding said API function calls to said second operating system;

user interface skeleton means for tracking the states and attributes of substantially all user interface objects created on the client computer;

a callback unit for synthesizing callback function calls to the callback routine in the application, said callback functions calls called ahead of their actual invocation by the API function on the client computer, said callback unit transmitting any return values from the callback routine to the client computer; and

a callback interceptor for receiving and intercepting callback calls from said second operating system, said callback interceptor for providing any return values to said second operating system in accordance with the return values received from the callback unit on the server computer.

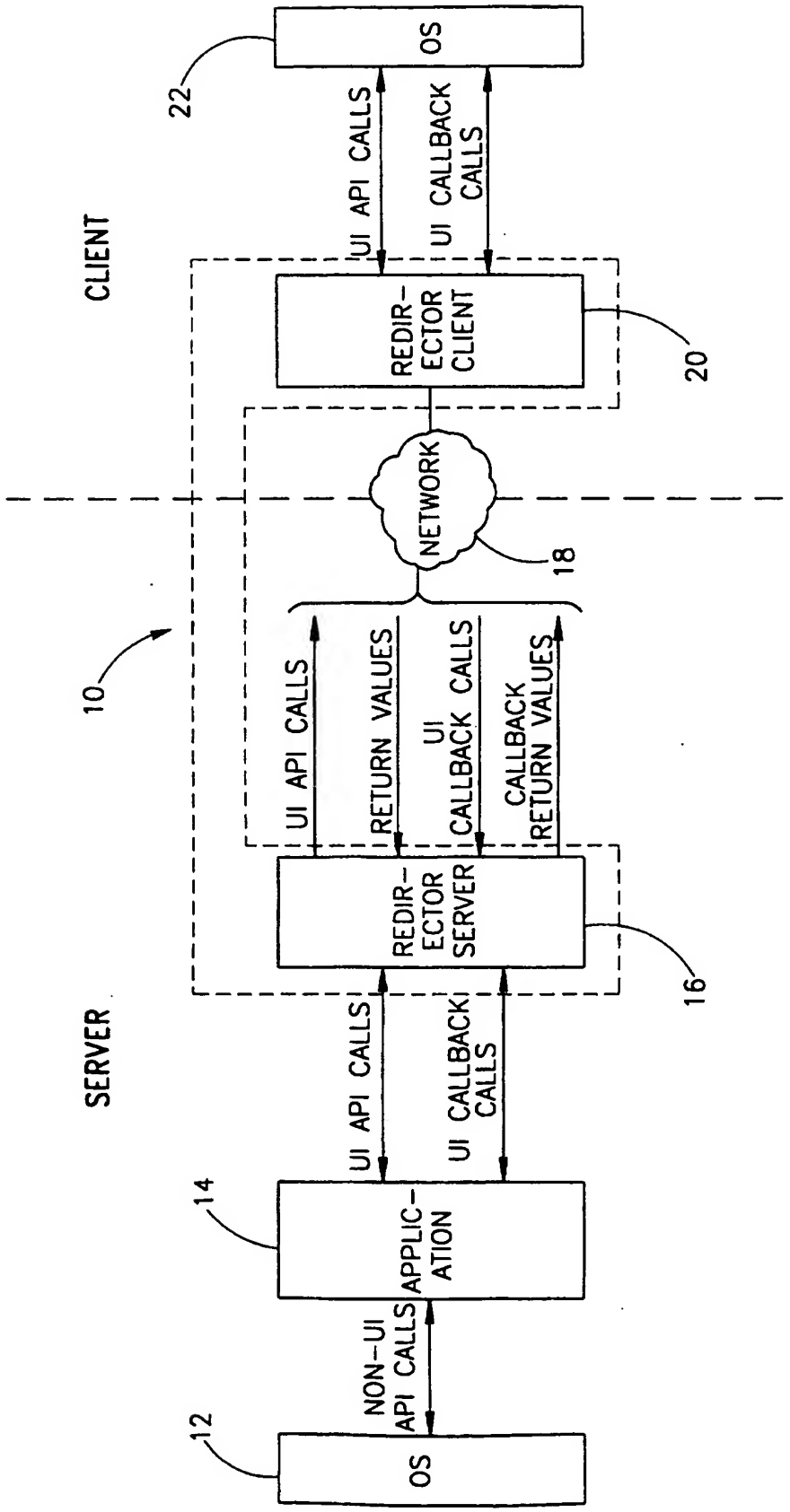

CLIENT
SERVER
10
22
OS
UI API CALLS
UI CALLBACK CALLS
REDIR-ECTOR CLIENT
20
NETWORK
18
UI API CALLS
RETURN VALUES
UI CALLBACK CALLS
CALLBACK RETURN VALUES
REDIR-ECTOR SERVER
16
UI API CALLS
UI CALLBACK CALLS
14
APPLIC-ATION
NON-UI API CALLS
12
OS

FIG. 1

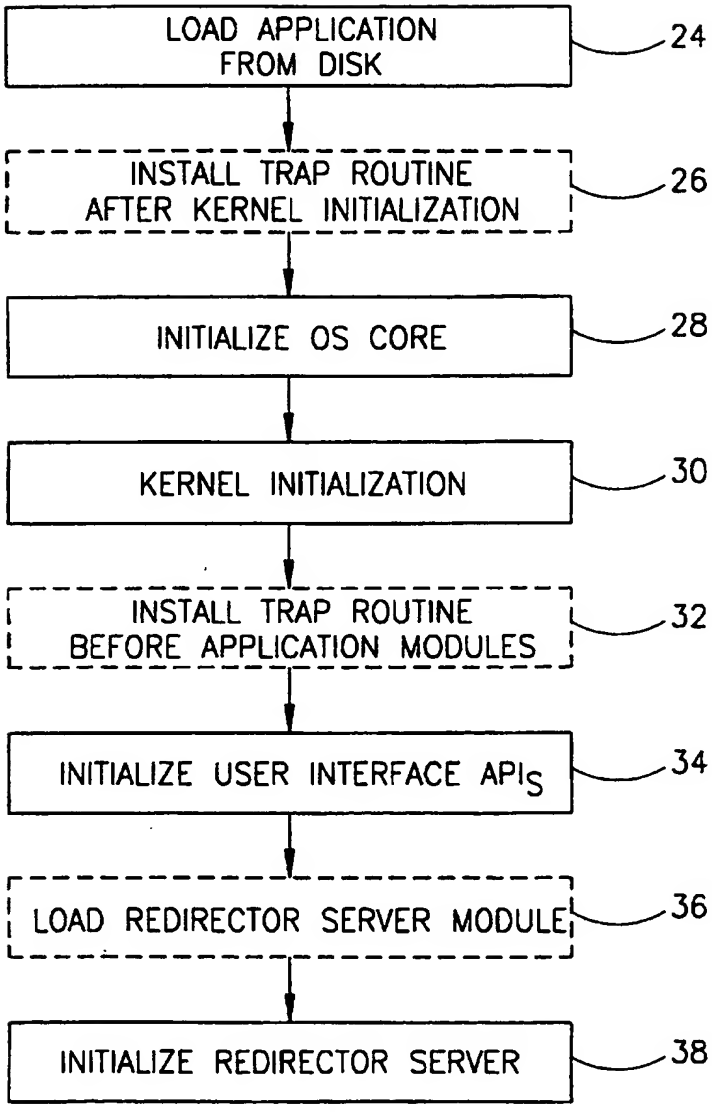

LOAD APPLICATION FROM DISK
24
INSTALL TRAP ROUTINE AFTER KERNEL INITIALIZATION
26
INITIALIZE OS CORE
28
KERNEL INITIALIZATION
30
INSTALL TRAP ROUTINE BEFORE APPLICATION MODULES
32
INITIALIZE USER INTERFACE APIS
34
LOAD REDIRECTOR SERVER MODULE
36
INITIALIZE REDIRECTOR SERVER
38

FIG.2

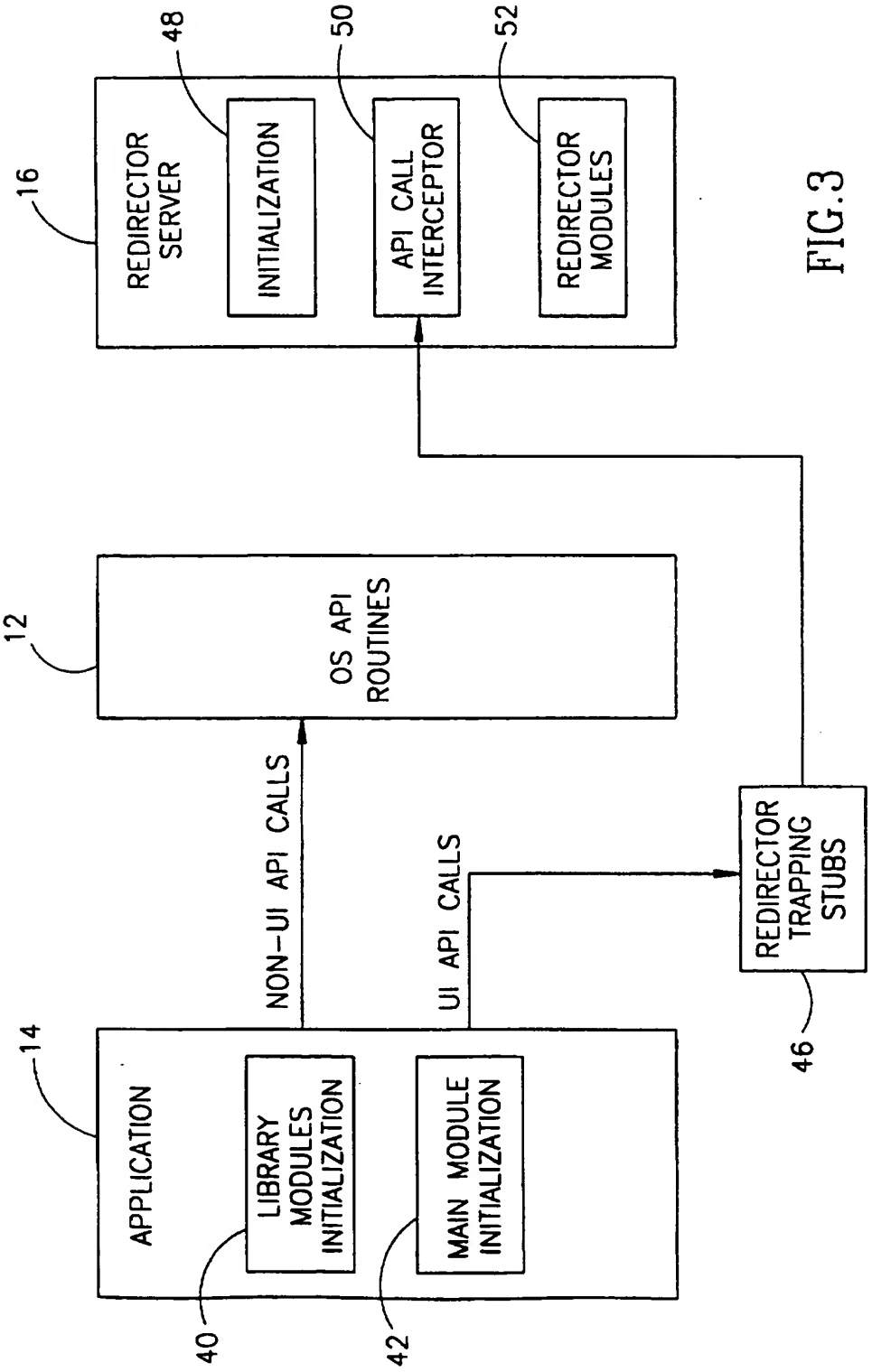

16
REDIRECTOR SERVER
48
INITIALIZATION
50
API CALL INTERCEPTOR
52
REDIRECTOR MODULES
12
OS API ROUTINES
NON-UI API CALLS
UI API CALLS
REDIRECTOR TRAPPING STUBS
46
14
APPLICATION
LIBRARY MODULES INITIALIZATION
MAIN MODULE INITIALIZATION
40
42

FIG.3

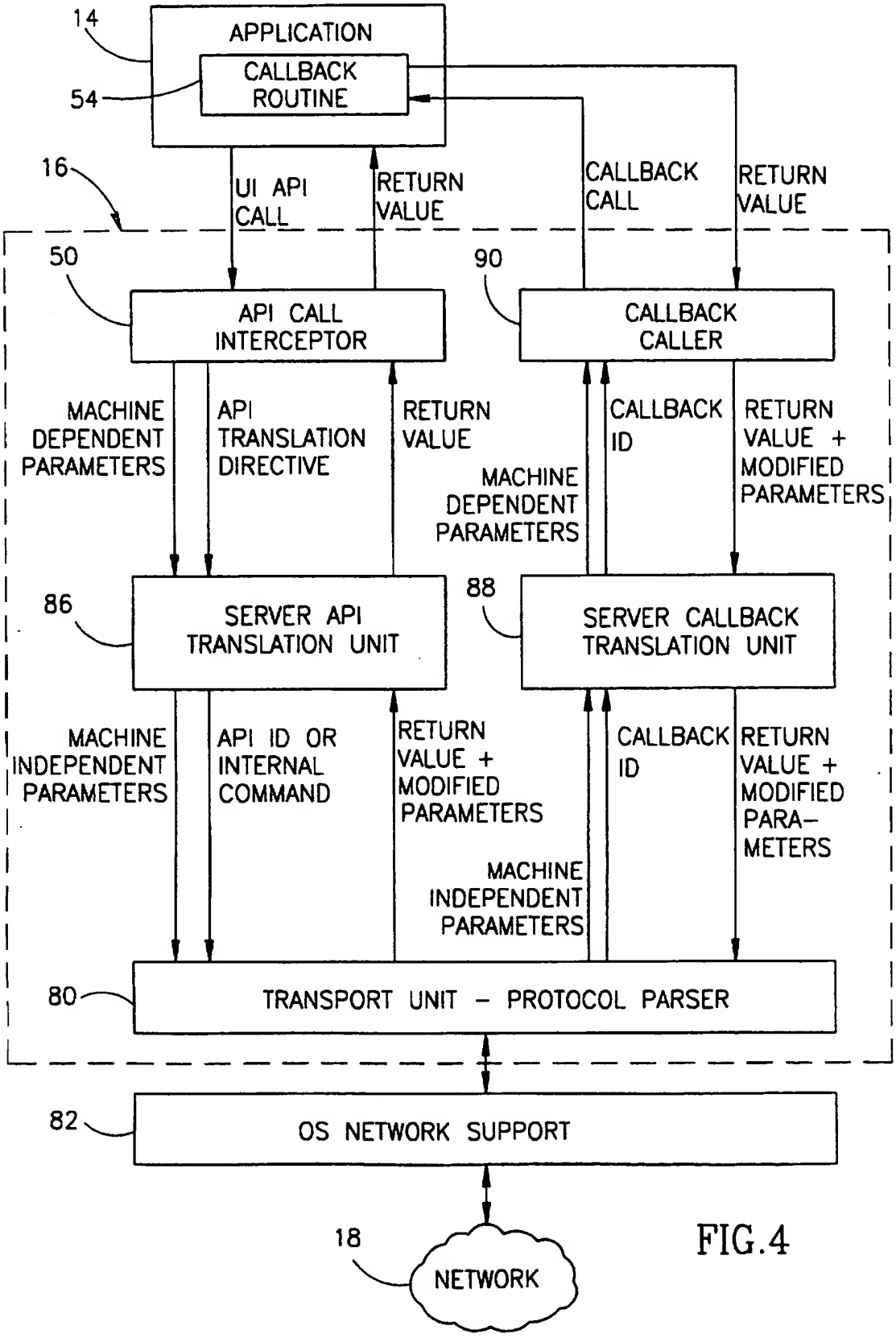

14
APPLICATION
54
CALLBACK ROUTINE
16
UI API CALL
RETURN VALUE
CALLBACK CALL
RETURN VALUE
50
API CALL INTERCEPTOR
90
CALLBACK CALLER
MACHINE DEPENDENT PARAMETERS
API TRANSLATION DIRECTIVE
RETURN VALUE
MACHINE DEPENDENT PARAMETERS
CALLBACK ID
RETURN VALUE + MODIFIED PARAMETERS
86
SERVER API TRANSLATION UNIT
88
SERVER CALLBACK TRANSLATION UNIT
MACHINE INDEPENDENT PARAMETERS
API ID OR INTERNAL COMMAND
RETURN VALUE + MODIFIED PARAMETERS
MACHINE INDEPENDENT PARAMETERS
CALLBACK ID
RETURN VALUE + MODIFIED PARA-METERS
80
TRANSPORT UNIT - PROTOCOL PARSER
82
OS NETWORK SUPPORT
18
NETWORK

FIG.4

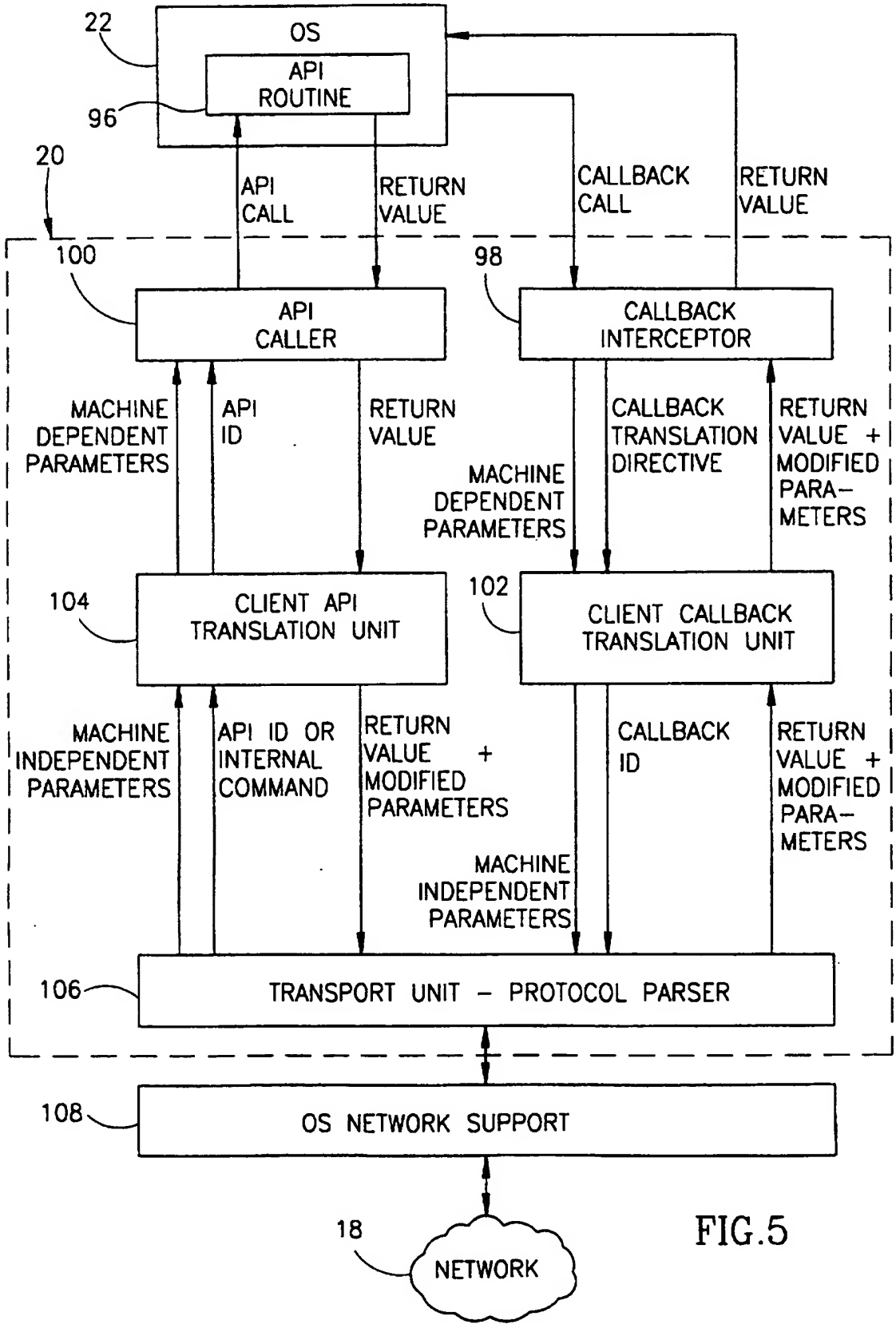

22
OS
API ROUTINE
96
20
API CALL
RETURN VALUE
CALLBACK CALL
RETURN VALUE
100
98
API CALLER
CALLBACK INTERCEPTOR
MACHINE DEPENDENT PARAMETERS
API ID
RETURN VALUE
MACHINE DEPENDENT PARAMETERS
CALLBACK TRANSLATION DIRECTIVE
RETURN VALUE + MODIFIED PARA-METERS
104
CLIENT API TRANSLATION UNIT
102
CLIENT CALLBACK TRANSLATION UNIT
MACHINE INDEPENDENT PARAMETERS
API ID OR INTERNAL COMMAND
RETURN VALUE + MODIFIED PARAMETERS
MACHINE INDEPENDENT PARAMETERS
CALLBACK ID
RETURN VALUE + MODIFIED PARA-METERS
106
TRANSPORT UNIT – PROTOCOL PARSER
108
OS NETWORK SUPPORT
18
NETWORK

FIG.5

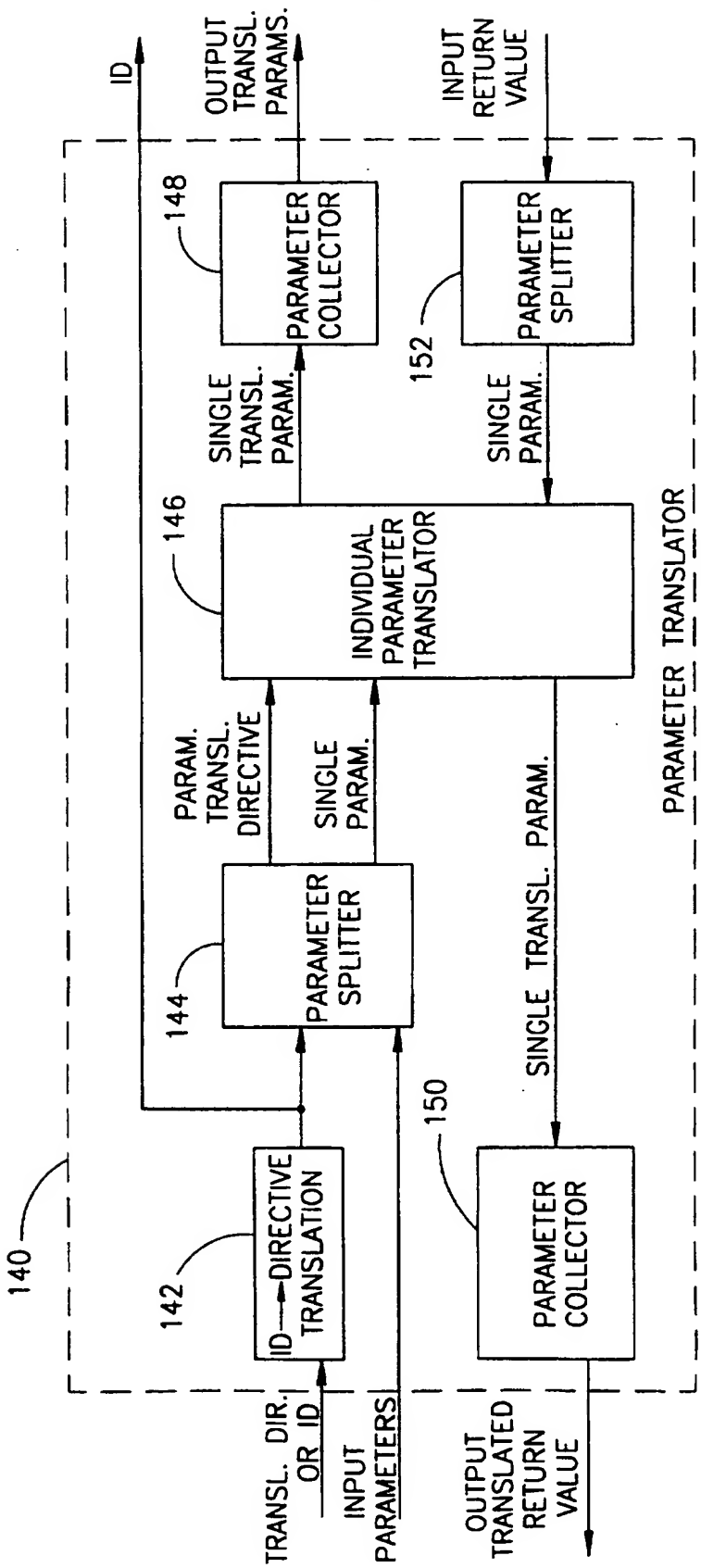

140
142
ID→DIRECTIVE TRANSLATION
144
PARAMETER SPLITTER
146
INDIVIDUAL PARAMETER TRANSLATOR
148
PARAMETER COLLECTOR
150
PARAMETER COLLECTOR
152
PARAMETER SPLITTER
ID
OUTPUT TRANSL. PARAMS.
INPUT RETURN VALUE
PARAM. TRANSL. DIRECTIVE
SINGLE PARAM.
SINGLE TRANSL. PARAM.
SINGLE TRANSL. PARAM.
SINGLE PARAM.
PARAMETER TRANSLATOR
TRANSL. DIR. OR ID
INPUT PARAMETERS
OUTPUT TRANSLATED RETURN VALUE

FIG.6

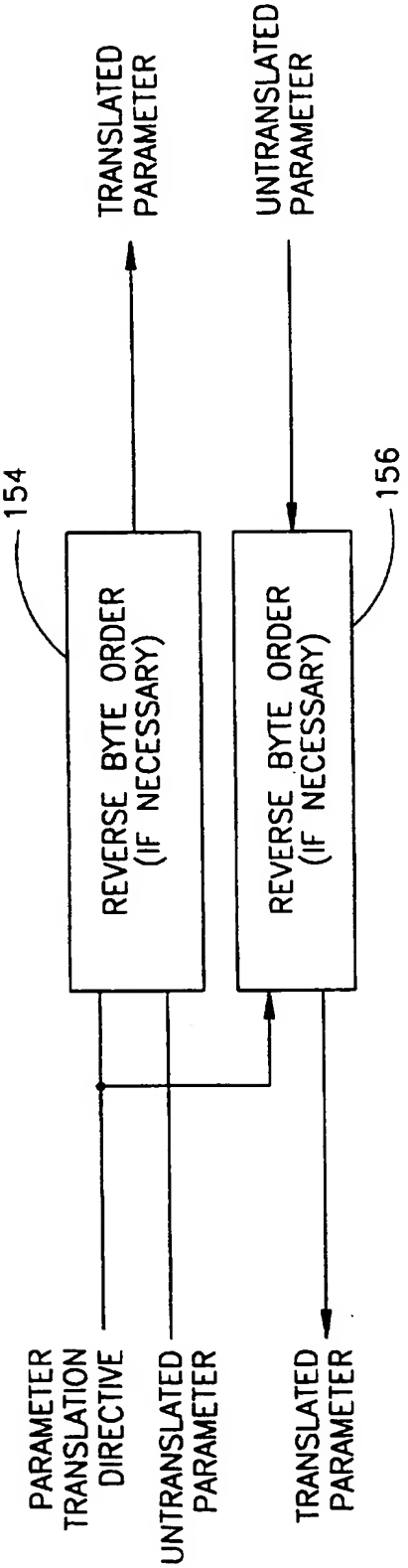

TRANSLATED PARAMETER
UNTRANSLATED PARAMETER
154
156
REVERSE BYTE ORDER (IF NECESSARY)
REVERSE BYTE ORDER (IF NECESSARY)
PARAMETER TRANSLATION DIRECTIVE
UNTRANSLATED PARAMETER
TRANSLATED PARAMETER

FIG.7

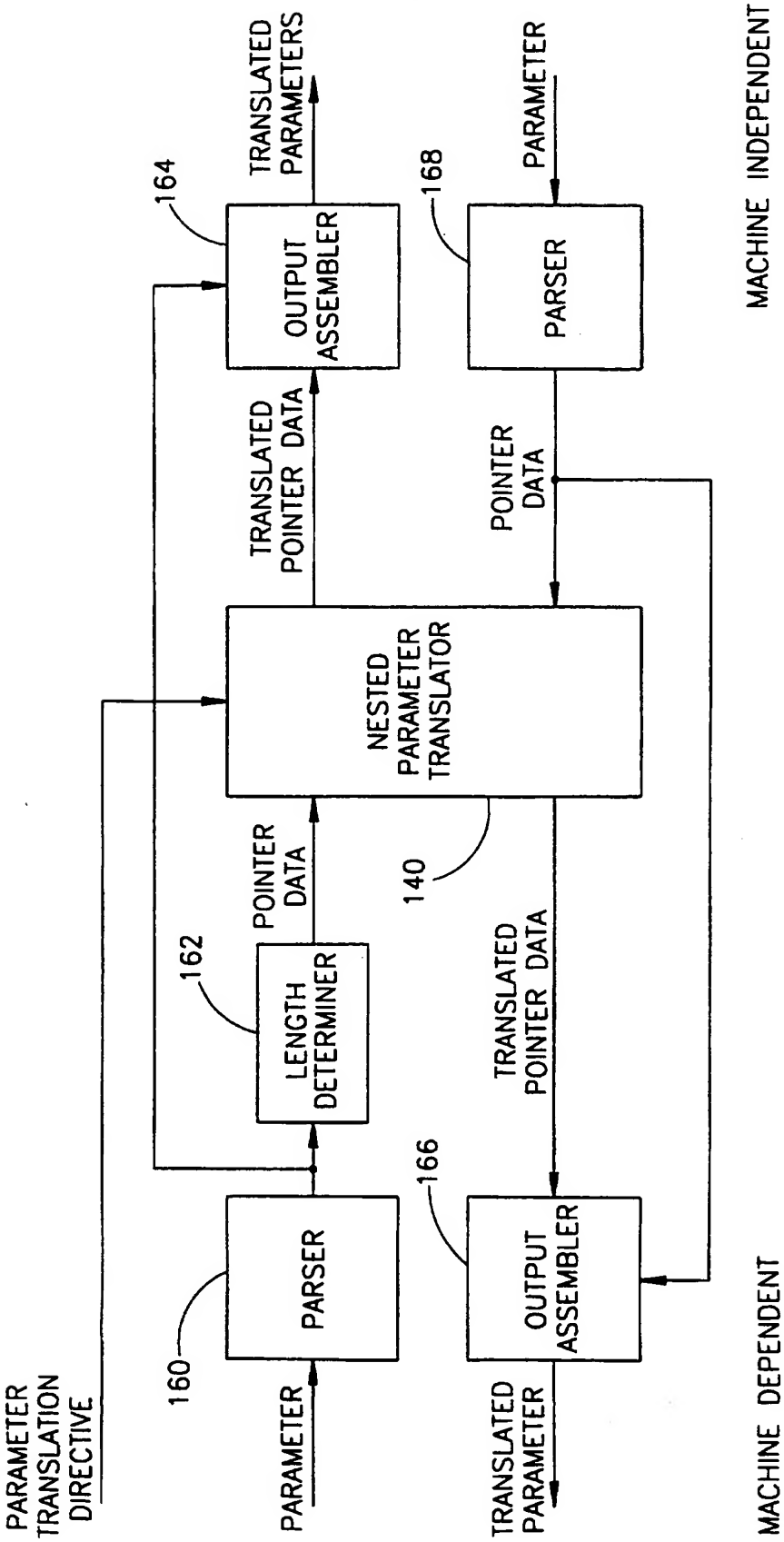

PARAMETER TRANSLATION DIRECTIVE
160
PARSER
PARAMETER
162
LENGTH DETERMINER
POINTER DATA
140
NESTED PARAMETER TRANSLATOR
TRANSLATED POINTER DATA
166
OUTPUT ASSEMBLER
TRANSLATED PARAMETER
MACHINE DEPENDENT
164
OUTPUT ASSEMBLER
TRANSLATED PARAMETERS
168
PARSER
PARAMETER
POINTER DATA
MACHINE INDEPENDENT

FIG.8

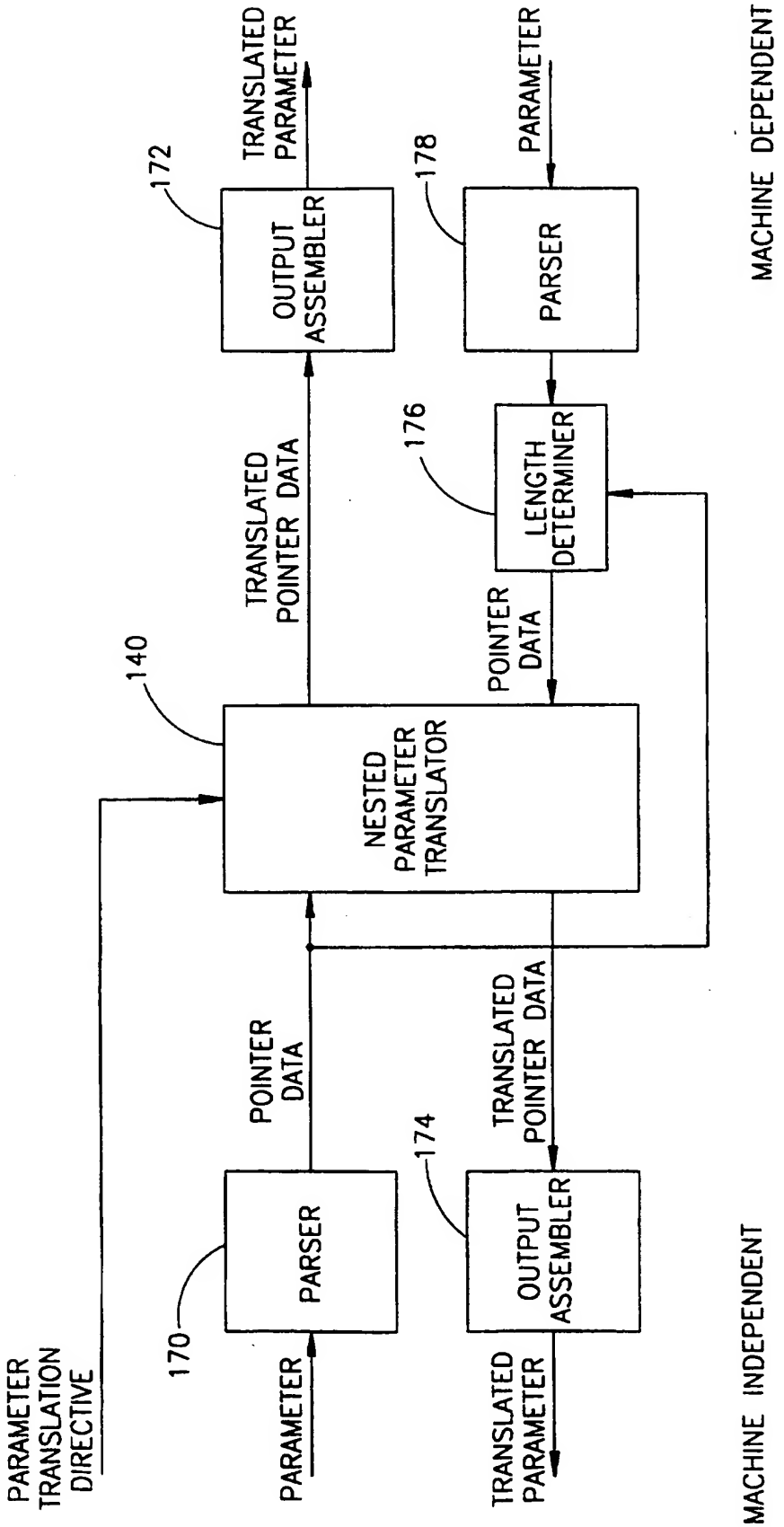

PARAMETER TRANSLATION DIRECTIVE
170
PARSER
PARAMETER
POINTER DATA
140
NESTED PARAMETER TRANSLATOR
TRANSLATED POINTER DATA
174
OUTPUT ASSEMBLER
TRANSLATED PARAMETER
MACHINE INDEPENDENT
176
LENGTH DETERMINER
POINTER DATA
TRANSLATED POINTER DATA
172
OUTPUT ASSEMBLER
TRANSLATED PARAMETER
178
PARSER
PARAMETER
MACHINE DEPENDENT

FIG.9

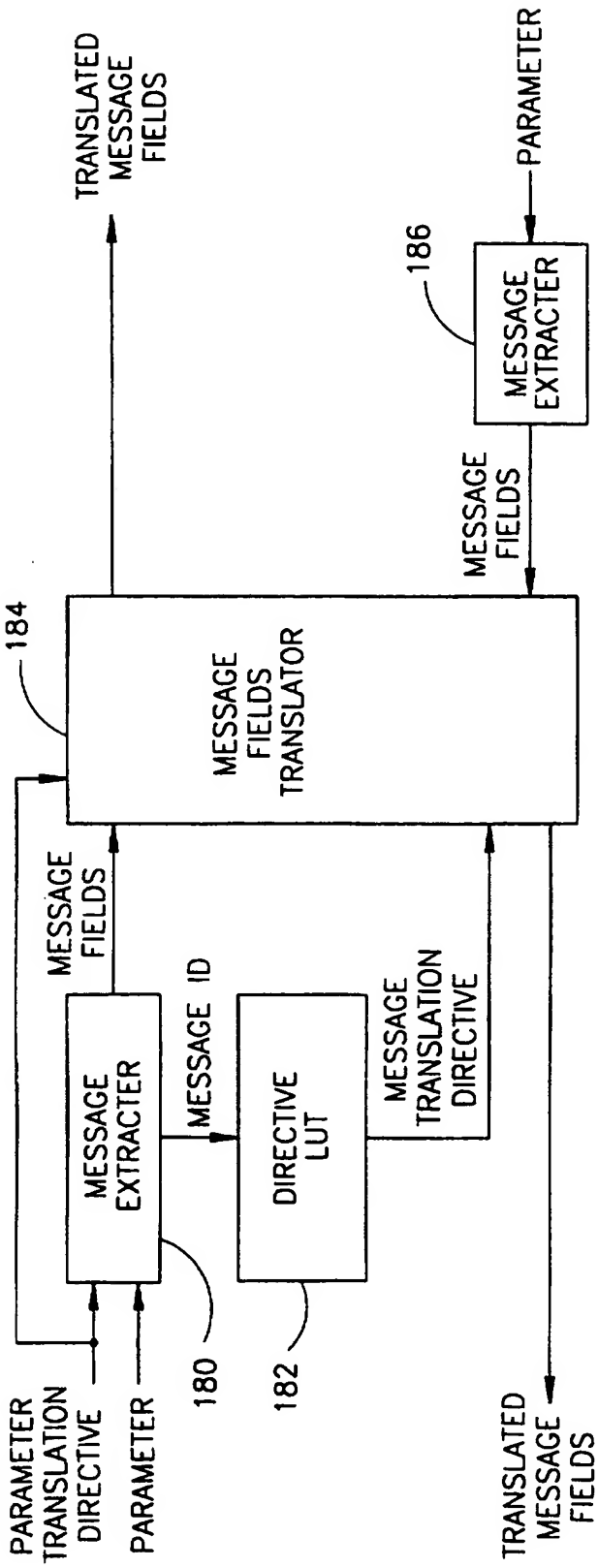

TRANSLATED MESSAGE FIELDS
PARAMETER
186
MESSAGE EXTRACTER
MESSAGE FIELDS
184
MESSAGE FIELDS TRANSLATOR
MESSAGE FIELDS
MESSAGE ID
MESSAGE TRANSLATION DIRECTIVE
MESSAGE EXTRACTER
DIRECTIVE LUT
PARAMETER TRANSLATION DIRECTIVE
PARAMETER
180
182
TRANSLATED MESSAGE FIELDS

FIG.10

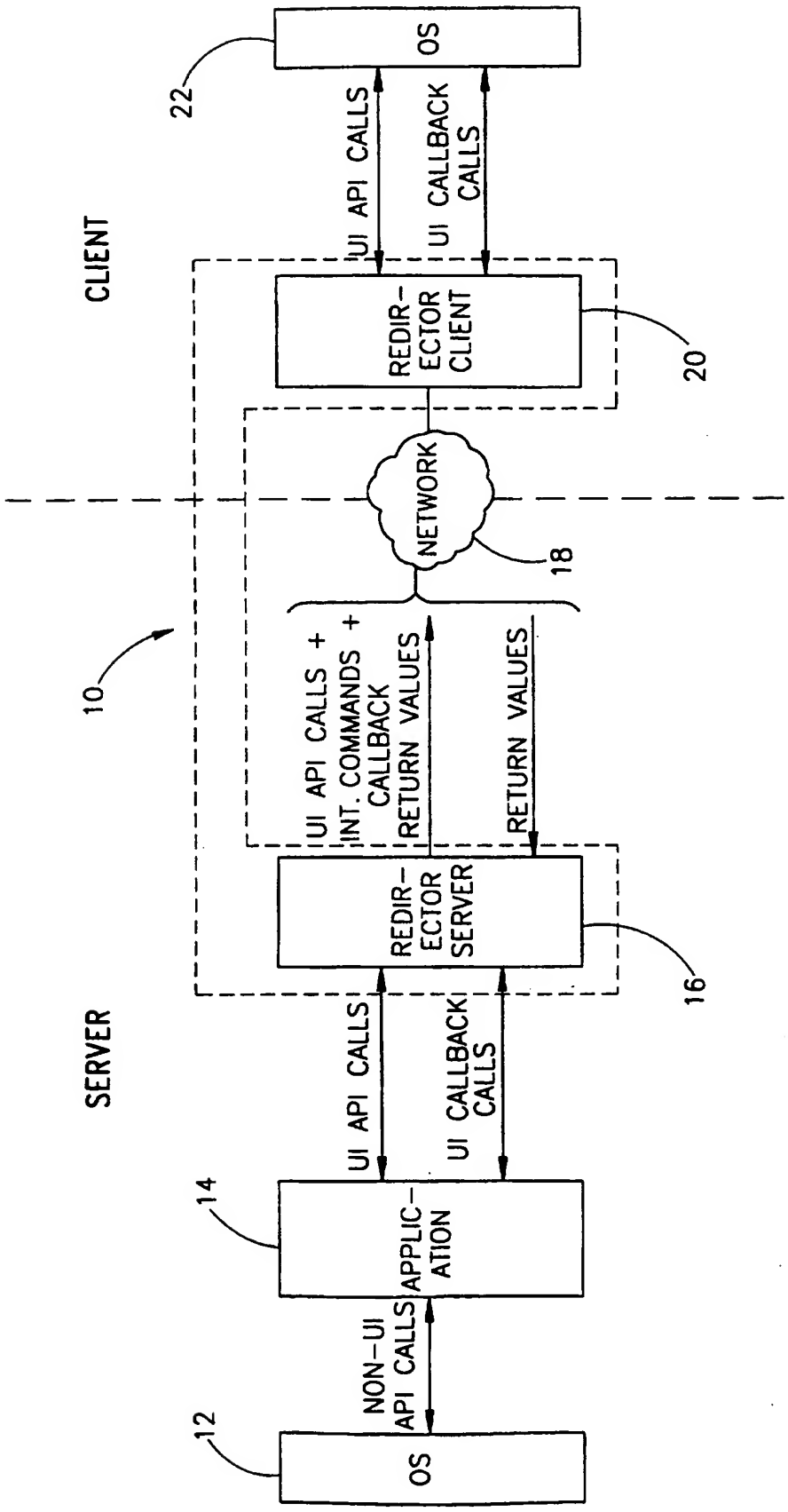

CLIENT
SERVER
10
22
OS
UI API CALLS
UI CALLBACK CALLS
REDIR-ECTOR CLIENT
20
NETWORK
18
UI API CALLS + INT. COMMANDS + CALLBACK RETURN VALUES
RETURN VALUES
REDIR-ECTOR SERVER
16
UI API CALLS
UI CALLBACK CALLS
14
APPLIC-ATION
NON-UI API CALLS
12
OS

FIG.11

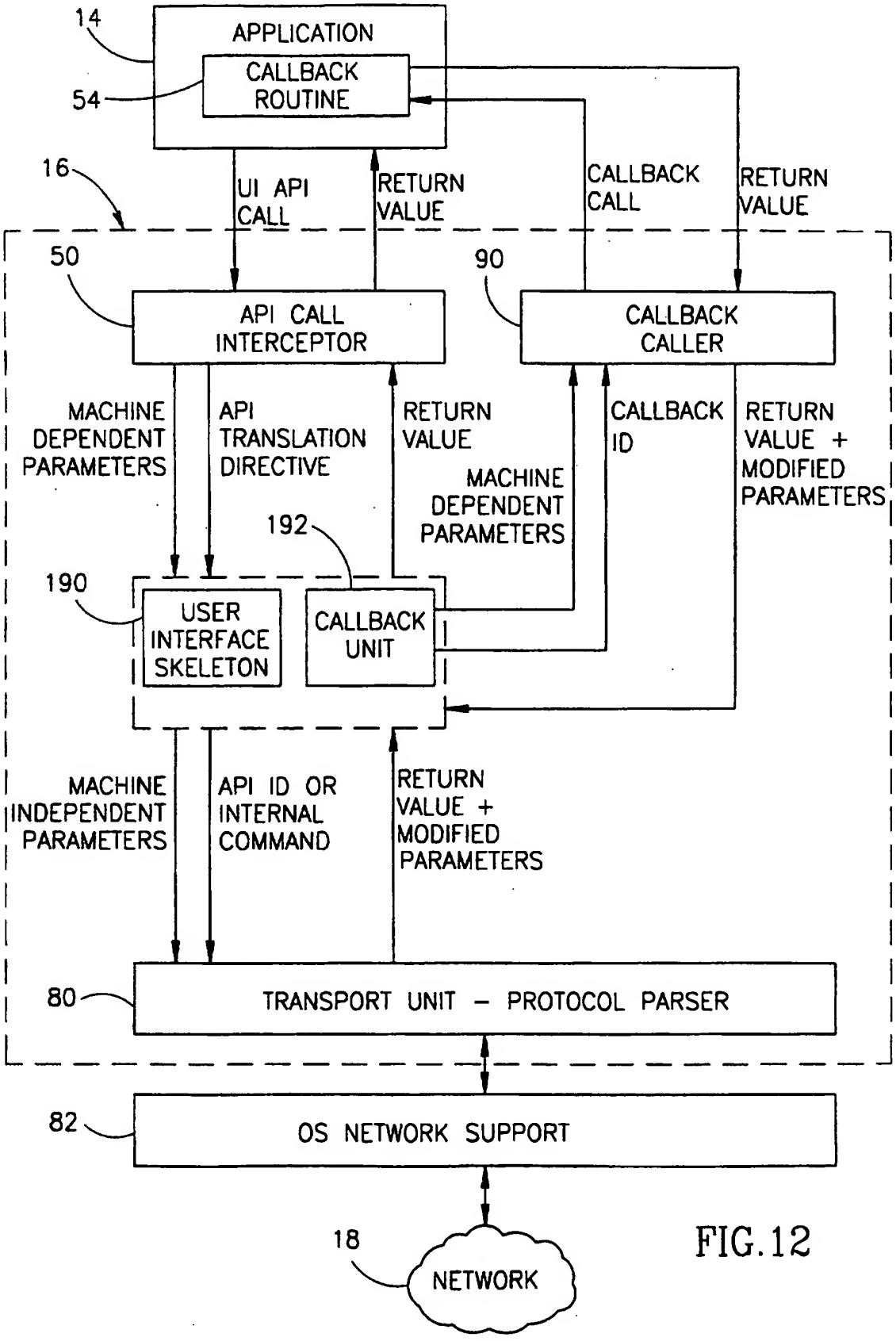

14
APPLICATION
54
CALLBACK ROUTINE
16
UI API CALL
RETURN VALUE
CALLBACK CALL
RETURN VALUE
50
API CALL INTERCEPTOR
90
CALLBACK CALLER
MACHINE DEPENDENT PARAMETERS
API TRANSLATION DIRECTIVE
RETURN VALUE
MACHINE DEPENDENT PARAMETERS
CALLBACK ID
RETURN VALUE + MODIFIED PARAMETERS
192
190
USER INTERFACE SKELETON
CALLBACK UNIT
MACHINE INDEPENDENT PARAMETERS
API ID OR INTERNAL COMMAND
RETURN VALUE + MODIFIED PARAMETERS
80
TRANSPORT UNIT – PROTOCOL PARSER
82
OS NETWORK SUPPORT
18
NETWORK

FIG.12

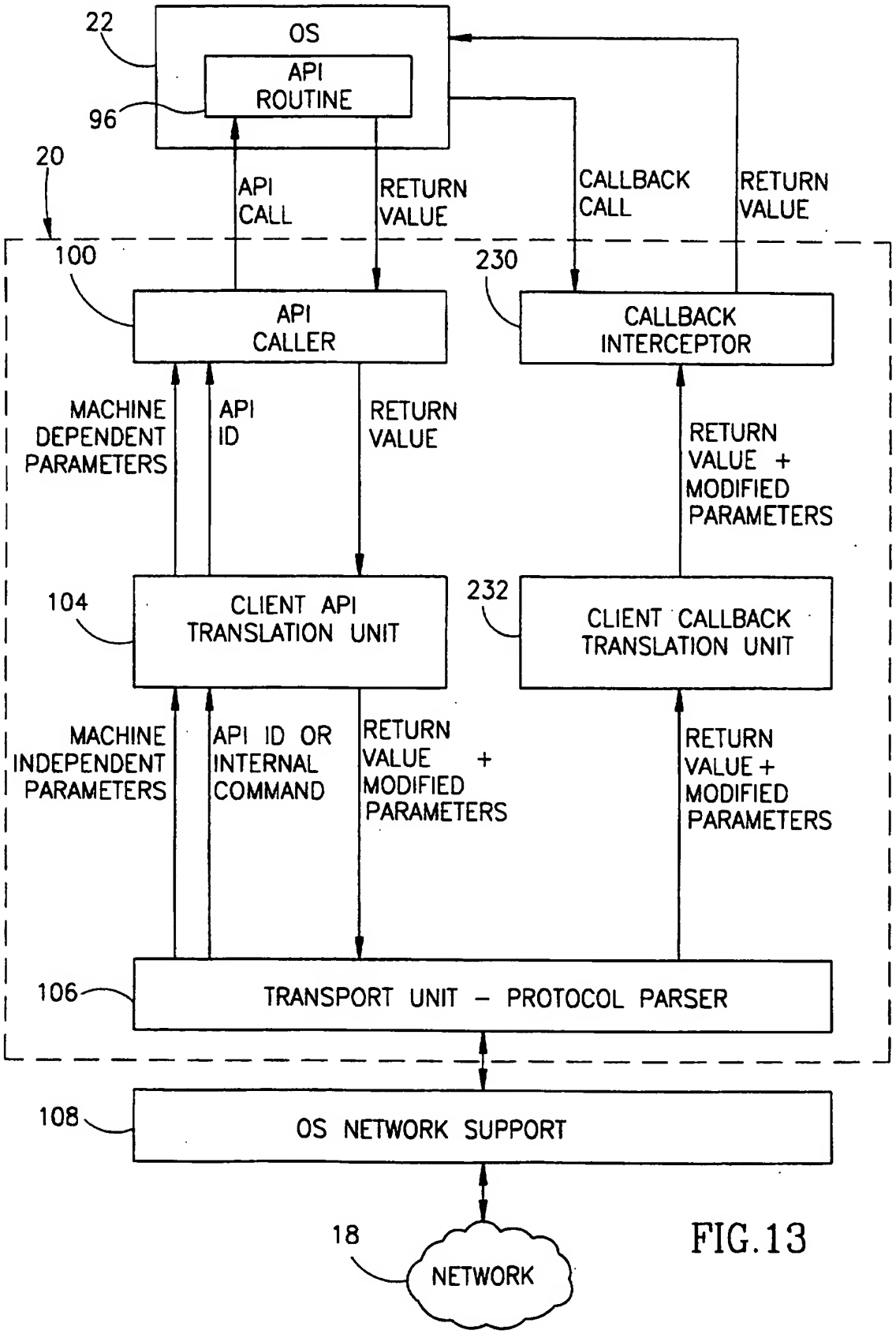

22
OS
API ROUTINE
96
20
API CALL
RETURN VALUE
CALLBACK CALL
RETURN VALUE
100
230
API CALLER
CALLBACK INTERCEPTOR
MACHINE DEPENDENT PARAMETERS
API ID
RETURN VALUE
RETURN VALUE + MODIFIED PARAMETERS
104
CLIENT API TRANSLATION UNIT
232
CLIENT CALLBACK TRANSLATION UNIT
MACHINE INDEPENDENT PARAMETERS
API ID OR INTERNAL COMMAND
RETURN VALUE + MODIFIED PARAMETERS
RETURN VALUE + MODIFIED PARAMETERS
106
TRANSPORT UNIT – PROTOCOL PARSER
108
OS NETWORK SUPPORT
18
NETWORK

FIG.13

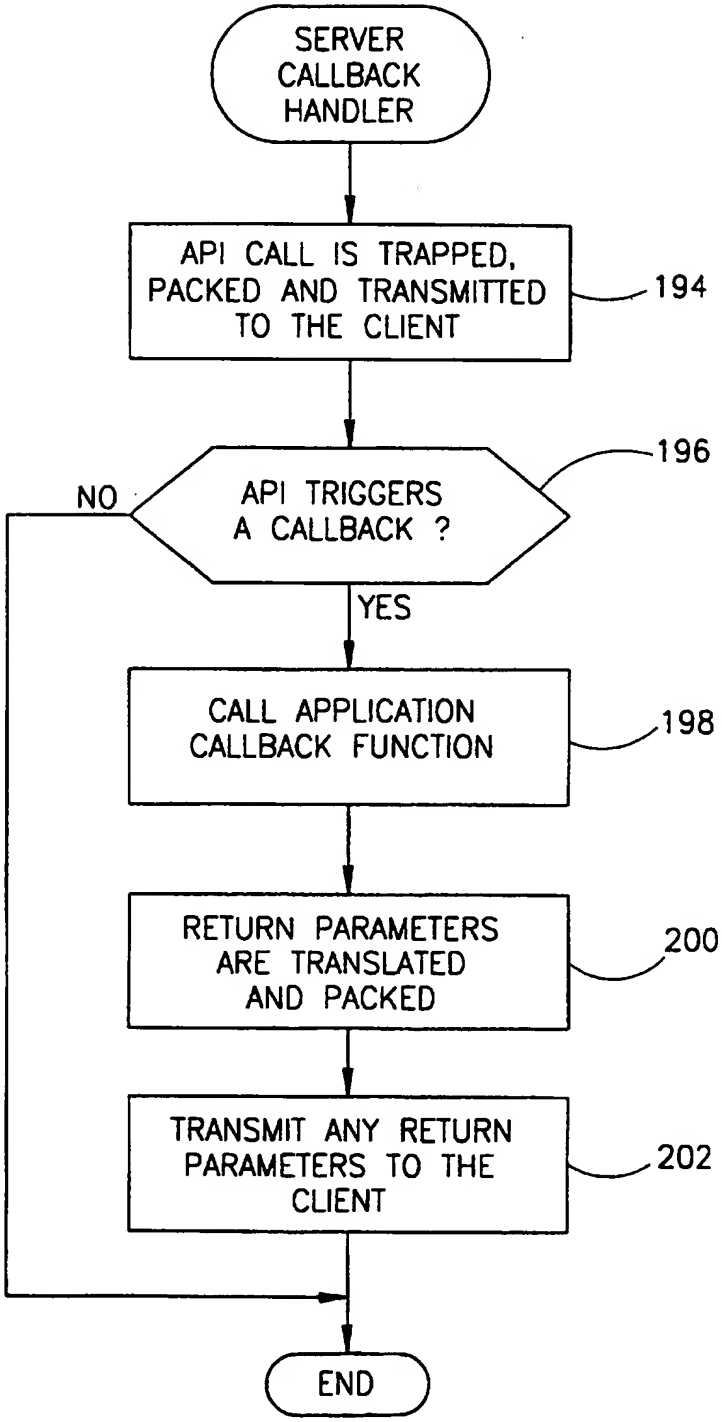

SERVER CALLBACK HANDLER
API CALL IS TRAPPED, PACKED AND TRANSMITTED TO THE CLIENT
194
API TRIGGERS A CALLBACK ?
196
NO
YES
CALL APPLICATION CALLBACK FUNCTION
198
RETURN PARAMETERS ARE TRANSLATED AND PACKED
200
TRANSMIT ANY RETURN PARAMETERS TO THE CLIENT
202
END

FIG.14

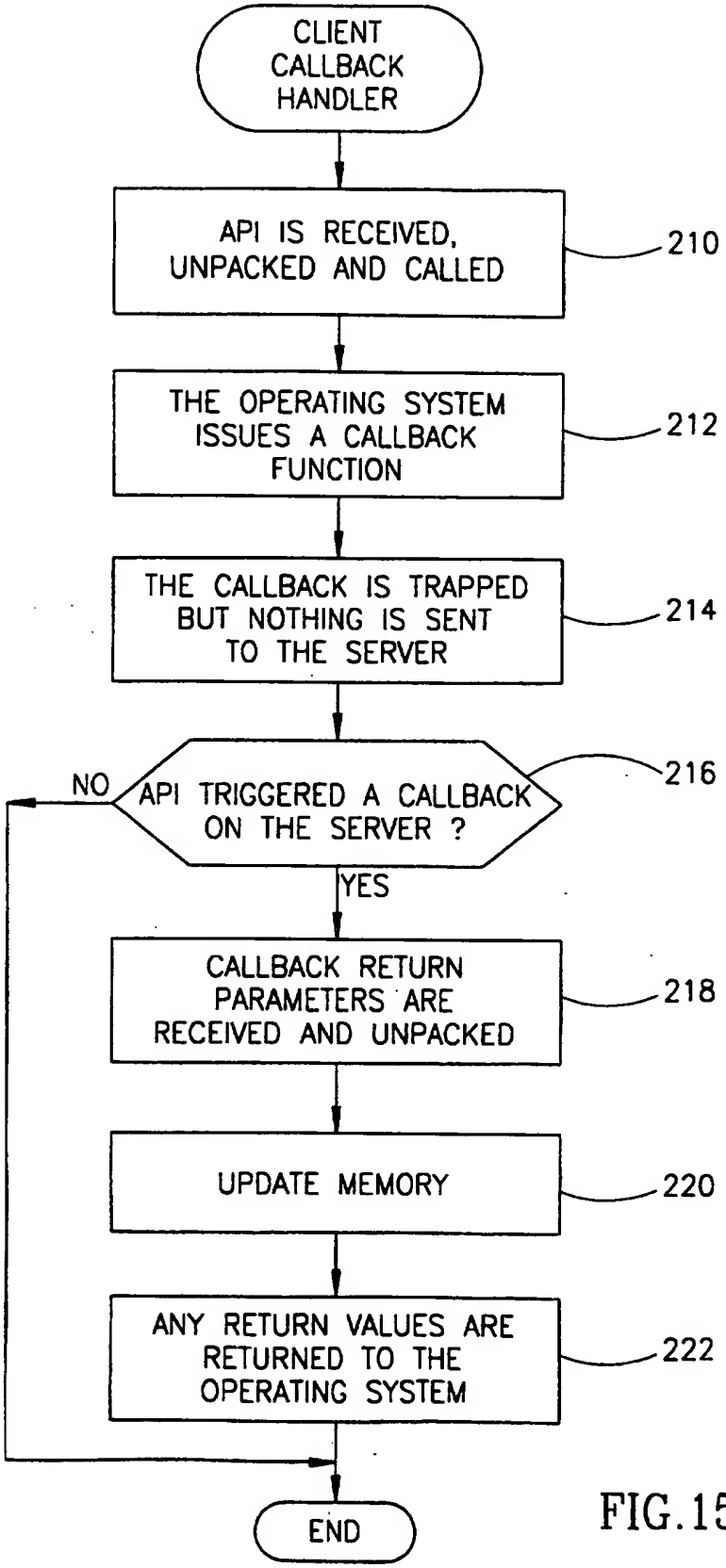

CLIENT CALLBACK HANDLER
API IS RECEIVED, UNPACKED AND CALLED
210
THE OPERATING SYSTEM ISSUES A CALLBACK FUNCTION
212
THE CALLBACK IS TRAPPED BUT NOTHING IS SENT TO THE SERVER
214
NO
API TRIGGERED A CALLBACK ON THE SERVER ?
216
YES
CALLBACK RETURN PARAMETERS ARE RECEIVED AND UNPACKED
218
UPDATE MEMORY
220
ANY RETURN VALUES ARE RETURNED TO THE OPERATING SYSTEM
222
END

FIG.15